NAVAL
POSTGRADUATE
SCHOOL

MONTEREY, CALIFORNIA

THESIS

STATISTICAL DEBUGGING

by

Toriano A. Murphy

March 2008

Thesis Advisor:	Mikhail Auguston
Second Reader:	Richard Riehle

REPORT DOCUMENTATION PAGE		*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.		
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** March 2008	**3. REPORT TYPE AND DATES COVERED** Master's Thesis
4. TITLE AND SUBTITLE Statistical Debugging		**5. FUNDING NUMBERS**
6. AUTHOR(S) Toriano Murphy		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000		**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A		**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.		
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited		**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

Software debugging is a time consuming and important step in the development and evolution of software systems. Debugging is a practice that normally gets the least praise but normally requires the most attention and effort. The aim of debugging is to find and reduce the number of faults in a program, thereby making a program behave the way it is expected. Even with the advances that have been made with computer speed, graphical user interfaces, networking abilities and storage capabilities the cost of debugging remains high.

The aim of this thesis is to build on the process of debugging using a statistical approach. Statistical debugging is not a new phenomenon, but a statistical debugging technique has been developed to assist in addressing the difficulties of isolating faults in software. The tool developed for debugging by this thesis will save time and effort in finding faults thereby saving money.

14. SUBJECT TERMS Statistical Debugging, Oracle			**15. NUMBER OF PAGES** 109
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

STATISTICAL DEBUGGING

Toriano A. Murphy
Lieutenant, United States Navy
B.S., Norfolk State University, 2001

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2008

Author:	Toriano Murphy
Approved by:	Mikhail Auguston Thesis Advisor
	Richard Riehle Second Reader
	Peter J. Denning Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Software debugging is a time consuming and important step in the development and evolution of software systems. Debugging is a practice that normally gets the least praise but normally requires the most attention and effort. The aim of debugging is to find and reduce the number of faults in a program, thereby making a program behave the way it is expected. Even with the advances that have been made with computer speed, graphical user interfaces, networking abilities and storage capabilities the cost of debugging remains high.

The aim of this thesis is to build on the process of debugging using a statistical approach. Statistical debugging is not a new phenomenon, but a statistical debugging technique has been developed to assist in addressing the difficulties of isolating faults in software. The tool developed for debugging by this thesis will save time and effort in finding faults thereby saving money.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

Section I describes the basic idea behind debugging and why it is such an important issue. The purpose and motivation behind this thesis is discussed as well.

Section II provides information on some of the work being conducted in the area of statistical debugging and provides directions on this thesis.

Section III details the preliminary work that achieved in order to build a solid case that statistical debugging is a good approach to debugging. Some initial investigation was done to build a greater understanding about statistical debugging.

Section IV talks about methodology used to design the instrumentation tool. All the decision making that went into the building the tool is discussed in this section

Section V presents the experiments and evaluation that was performed on the instrumentation tool. The input data was randomly generated and therefore is not shown, the results are shown.

Section VI presents the conclusion of this thesis and provides some recommendations for future work.

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank Dr. Mikhail Auguston and Professor Richard Riehle for providing the direction and the inspiration for this thesis. I would like to thank Professor Kevin Squire for introducing me to the concept of statistical debugging. I would like to give a special thanks to Dr. Auguston for all the countless hours that you dedicated to keeping my compass on true north.

I would like to thank my wife, Audrey, and daughter, Chelsea, for giving me time to focus on this research. Last, but certainly not less, I must thank God by whom all this is possible.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

Over the last few decades software size and complexity has grown at an alarming rate. Methods for testing and finding bugs in programs have not kept pace with the growth rate of software growth. In a lot of cases, debugging is still performed by humans with no assistance from software tools.

> Debugging is the dirty little secret of computer science. Despite the progress we have made in the past 30 years, faster computers, networking, easy-to-use graphical interfaces, and everything else, we still face some embarrassing facts about software development. First, computer programs often don't work as they should, making software development costly. And too much buggy software reaches end users, leading to needless expense and frustration. That's unfortunate, but most surprising is the fact that when something does go wrong, the people who write the programs still lack good ways of figuring out exactly what went wrong. Debugging is still, as it was 30 years ago, largely a matter of trial and error. Bordering on scandal is the fact that the computer science community as a whole has largely ignored the debugging problem. This is inexcusable, considering the vast economic cost of debugging and the emotional toll buggy software takes on users and programmers. Today's commercial programming environments provide debugging tools that are little better than the tools that came with programming environments 30 years ago [7].

The aim of this thesis is to address the difficulty of testing and debugging by displaying an instrumentation tool and presenting a methodology for finding bugs in software. The instrumentation tool was written in the Java programming language. There are many programming languages that could have been utilized to write the instrumentation tool more efficiently but Java was utilized because it is a mainstream language. The instrumentation tool reads in a *target program*. The instrumentation tool inserts code around three different types of statements in the target program, "if statements", "while statements" and "for statements". These statements are being monitored because these are critical decision making points in a program which always yield a true or false outcome and provide useful information when viewing them as a program profile.

The main conjecture underlying this research is that the number of True/False predicate outcomes in a program for correct and incorrect execution should be different. Of course, these types of errors do not include all possible errors that could occur in the program, but a significant part of them can be pinpointed using the following approach. An *oracle* is a program or snippet of code that is utilized to check the results of the *target program* which may have errors of commission inside it. In the future this *target program* will be referred to as the *test program*. The *test program* and the *oracle* are executed using the same random input data. The results of each predicate after execution in both programs will be recorded in a "Counter Class" for later evaluation. The *oracle* will be used as a reference point for the predicate profile in the *test program*. The delta is defined as the absolute difference between the total numbers of true visits divided by the total number of visits in correct and incorrect executions of the test program.

It is important to understand that there are several steps to debugging a program. First the fault in a program must be isolated; next the cause of the fault must be understood. The cause of the fault may involve other parts of the program. Next a solution has to be derived to address the problem, last the solution has to be applied, and then tested.

In this research, the emphasis is placed on isolating the fault since this is normally the most time consuming. Normally iterative testing is required to isolate the fault, meaning it will take more time, which ultimately means more money. Some amount of time must be spent testing and debugging, the goal of debugging is to find the greatest amount of faults with the least amount of time and effort.

B. PROBLEM STATEMENT AND MOTIVATION

The purpose of this research is to investigate the plausibility of a proposed statistical debugging technique to isolate faults in a program with some degree of certainty.

First, it must be reiterated that software debugging is a difficult and time consuming task. An instrumentation tool has been designed to help in the quest to find faults in a program. The instrumentation tool will instrument target programs so that a comparison can be made about the predicate profiles in a correct and incorrect test program execution.

In this research an oracle will be utilized to verify that the results of a test program are correct. It is important to understand that the oracle may be partially in-complete, even in these cases the results will still be of assistance in finding faults in a program.

The primary motivation for this research is to show that this statistical debugging technique is able to find faults in a program. Focusing more research on this topic in the future will increase the amount of time that can be saved during testing and debugging. Through experimentation with several test cases in this research, data will be generated to express the success of this instrumentation tool and to point out areas for future research.

Even though statistical debugging is not exhaustive (nor can it be) this instrumentation tool could be used to aid in the testing and debugging process. It can be used to save precious time and resources through the use of the automated instrumentation tool presented in this paper.

Increased software size and the rate at which software is evolving demands that a more efficient method of debugging be created to keep pace with these increasing demands.

THIS PAGE INTENTIONALLY LEFT BLANK

II. RELATED WORK

Statistical debugging is a relatively new approach to debugging software. There are just a few statistical debugging methodologies and techniques that are currently in use today. The goal of this section is to discuss some of these techniques and to address some of the strengths and weaknesses associated with each technique.

A. STATISTICAL DEBUGGING: A HYPOTHESIS TESTING-BASED APPROACH

SOBER is an approach that looks at two types of fault localization techniques to find faults in software. First, the static localization technique compares the source code to a program model. Second, the dynamic localization techniques which contrast correct versus incorrect program runs.

> Unlike existing statistical approaches that select predicates correlated with program failure, SOBER models the predicate evaluation in both correct and incorrect executions and regards a predicate as fault-relevant if its evaluation pattern in incorrect executions significantly diverges from that in correct ones. Featuring a rationale similar to that of hypothesis testing, SOBER quantifies the fault relevance of each predicate in a principled way [9].

The SOBER methods takes an interesting approach to finding bugs, instead of analyzing the delta between individual predicates in a correct and incorrect program, it makes the attempt to find faults by "quantifying the divergence between the models of correct and incorrect program executions" [9].

A major challenge with debugging is that faults are hard to isolate and can be subtle within a program. SOBER addresses the problem of what is referred to as the "imperfect world" problem by utilizing a test suite that is fault free as the test oracle to say if the each execution of a model is passing or failing. SOBER has an advantage over other statistical debugging techniques in that it makes the claim of being able to localize software faults without knowing anything in advance of the programs semantics. The disadvantage with SOBER is that it is an empirical study and there is a risk taken in accepting the validity of the experimental results. This is the case with any empirical study and it must be taken into consideration.

B. FAULT-AWARE FINGERPRINTING: TOWARDS MUTUALISM BETWEEN FAILURE INVESTIGATION AND STATISTICAL DEBUGGING

This paper takes an interesting approach. The author takes two concepts that have been used to find faults individually and ties them together. Failure investigation and statistical debugging had always been studied separately until now. The goal is to combine the two to assist in finding faults in collected program failing traces. Most complex programs like Microsoft Windows have a failure reporting feature which can send failing traces of the program back to Microsoft for analysis, the goal of this paper is to prove that these failing trace can be used for more than just conducting analysis, they can be used to diagnosis faults.

> Failure investigation and statistical debugging are two critical components in automated failure analysis. Although related, the two components have always been studied separately in previous work. In this paper, we propose a fault-aware fingerprinting technique that relates the two components and puts them into mutualism, i.e., each component benefits and benefits from the other. Technically, we use statistical debugging algorithms to find the fault location each failing trace suggest, and take the suggested fault location as the fingerprint of the failing trace. The fingerprint is *fault-aware* in that it embodies the fault location each failing trace suggests. With fingerprinting, we can cluster together failing traces that suggest similar fault locations, rather than those that are literally similar to each other as done by previous work. We observed from experiments that the fingerprint-based clustering renders more meaning results than the literal trace similarity-based approach. On the other hand, we also observe that the investigation of failing traces in the fingerprint space can help developers interpret statistical debugging results [5].

The paper is a follow on to the previous work done with the SOBER algorithm (mentioned in the previous section) and it combines the fault-aware fingerprinting technique. The fingerprints which represent a failing trace is said to embody the fault that is pointed to by each failure trace, with that said an election method is used to find the faults. This is accomplished by treating each predicate which has been instrumented as a candidate; the citizens who can vote in this case are the failing traces.

This approach would be even stronger if all predicates where instrumented in the test program, from there address the problem of rank aggregation which comes into play when consolidating a number of rankings.

This paper presents a great question "we are unclear what future work is really worthwhile and would like to know how industry processes the huge volume of collected failing traces" [5].

C. SCALABLE STATISTICAL BUG ISOLATION

The aim of this paper is to address statistical debugging by presenting an algorithm that can single out faults in a program.

> We present a statistical debugging algorithm that isolate bugs in programs containing multiple undiagnosed bugs. Earlier statistical algorithms that focus solely on identifying predicators that correlate with program failure perform poorly when there are multiple bugs. Our new technique separates the effects of different bugs and identifies predicators that are associated with individual bugs. These predicators reveal both the circumstances under which bugs occur as well as the frequencies of failure modes, making it easier to prioritize debugging efforts. Our algorithm is validated using several case studies, including examples in which the algorithm identified previously unknown, significant crashing bugs in widely used systems [2].

This paper builds off of previous work that initially focused on programs with a single bug. This paper sets some of the preliminary ground work for the direction of this thesis because it addresses the basis for all statistical debugging. Most methods of statistical debugging monitor a subset of the predicates in a program using some type of monitoring technique. The research in this paper builds on these basic concepts used in *Scalable Statistical Bug Isolation* paper [2]. An instrumentation tool was designed to instrument a program and to apply a statistical debugging technique which will be analyzed in detail in Chapter V. The goal is to see how successful this tool is at isolating faults. Instead of only monitoring only a subset of the predicates in a program, all predicates will be monitored with the exception of the case statement. The oracle that will be used to check result will be a copy of the program being tested with no faults introduced in the program.

III. PRELIMINARY WORK

A. STARTING POINT

In order to break the inertia and get a glimpse of what statistical debugging is and may have to offer to the testing and debugging aspect of software development, some reference point had to be gained. Before any code was ever run on a computer, a few hand written algorithms were used to see what the preliminary outcome would be, this was accomplished by using a few toy (small) programs.

The goal of statistical debugging as defined in the introduction is to find faults in a program by identifying the predicates with the greatest deltas between the predicates in a correct and incorrect program execution. The hope is that the predicates with the highest percentage delta between the two programs will point to or within the vicinity of the fault in the *test program*. The advantage point in this research is that an *oracle* exists and it will say when the *test program* is giving correct or incorrect results during runtime. In order for statistical debugging to work, there must be something available to say when the results of a *test program* are correct or incorrect. The *oracle* may be a partial one, but this is still adequate and better than not having one at all. Even a partial *oracle* helps in finding out how accurate this statistical debugging technique is at predicting faults on the basis of monitoring predicate behavior in a program execution.

Correct program

```
int x1 = 0;        int x2 =0;        int  x3 = 0;
int y1 = 0;       int y2 = 0;        int y3 = 0;
int z = 0;

if (x1 ≤ x2)     then    y1 = x1; y2 = x2;
                 else     y1 = x2; y2 = x1;

if (y2 ≤ x3)     then    y3 = x3;
                 else    y3 = y2; y2 = x3;

if (y1 ≤ y2)     then    z = y2;
                 else    z = y1;

print z;
```

Figure 1. Hand written algorithm.

Simple programs like the one in Figure 1 were utilized to conduct preliminary experimentation. This program was written to return the middle number when given three numbers as input. The program has three conditional statements (or predicates). The variables that will be used as input are represented by x1, x2 and x3. The temporary variables in the program are y1, y2 and y3. Three different experiments where observed in this program. Each experiment is conducted by introducing an error into the simple program and running six test cases for each experiment.

Test program

```
int x1 = 0;        int x2 =0;        int  x3 = 0;
int y1 = 0;       int y2 = 0;        int y3 = 0;
int z = 0;

if (x1 ≤ x2)      then     y1 = x1; y2 = x2;
                  else     y1 = x2; y2 = x1;

if (y2 ≤ x3)      then     y1 = x3;
                  else     y3 = y2; y2 = x3;

if (y1 ≤ y2)      then     z = y2;
                  else     z = y1;
```

Figure 2. Case 1 Test Program.

A correct version of the program, which we will refer to as the *oracle,* the *test program* which has an error of commission can be seen in Figure 2. The error of commission was inserted directly after the second if statement (y2 < x3) and is placed in bold letters.

Input used for correct and incorrect programs implementing "branch coverage criterion"

			Oracle	Test
1) x1=1	x2=3	x3=5	z = 3	z = 3
2) x1=3	x2=1	x3=5	z = 3	z = 3
3) x1=5	x2=1	x3=3	**z = 3**	**z = 5**
4) x1=1	x2=5	x3=3	**z = 3**	**z = 5**
5) x1=3	x2=5	x3=1	**z = 3**	**z = 5**
6) x1=5	x2=3	x3=1	**z = 3**	**z = 5**

Figure 3. Input used in Case 1.

Figure 3 shows the input that was used in the *oracle* and *test program.* A branch coverage criterion was utilized to see what the results would look like if all possibly paths were utilized. The results of z in case 1 seems to be closely related to the order of the input. All the input sets that had variable x3 as the largest integer in the set, the oracle and the test program have identical results. In the input set where x3 is not the largest the resulting z which can be seen in Figure 3 are all different.

Correct predicate profiles results	**In-correct predicate profiles results**
1) 11**1**	1) 11**0**
2) 011	2) 011
3) 001	3) 001
4) 1**01**	4) 1**10**
5) 100	5) 100
6) 000	6) 000

Figure 4. Case 1, profile results.

Figure 4 shows the predicate profile results for both the *oracle* and *test program* based on the input that was utilized in Figure 3.

The three bit sequence when viewed from left to right represents the result of each predicate after execution. There are several things that must be considered when observing the results. A true predicate outcome will be represented by the number one and a false outcome will be represented by the number zero.

In case 1, out of the six different sets of input only two sets of input rendered results with different outcome. This is interesting because in the four results that are the same, the *test program* provided the same results as the *oracle*. Four out of six inputs rendered a false positive output.

Second, the two instances where the results are different, the only difference lies in the last bit. This is interesting, because in both the *oracle* and the *test program* the program is identical, no faults are introduced until the second predicate. The code that was changed in the *test program* was the code directly following the second predicate. This gives some perspective that the logical flow of the *target program* is very important.

The relationship or closeness of a fault to the predicate giving indication of a fault can be correlated to where the fault resides with respect to the logical flow of the program.

<u>Test program</u>

```
int x1 = 0;      int x2 =0;      int  x3 = 0;
int y1 = 0;     int y2 = 0;      int y3 = 0;
int z = 0;

if (x1 ≤ x2)    then   y1 = x2; y2 = x1;
                else    y1 = x2; y2 = x1;

if (y2 ≤ x3)    then    y3 = x3;
                else    y3 = y2; y2 = x3;

if (y1 ≤ y2)    then    z = y2;
                else    z = y1;
```

Figure 5.   Case 2 Test Program.

Figure 5 shows the test program utilized in case 2 with the error of commission displayed in bold letters.

Correct predicate profiles results	**In-correct predicate profiles results**
1) 11**1**	1) 11**0**
2) 011	2) 011
3) 001	3) 001
4) 1**01**	4) 1**10**
5) 100	5) 100
6) 000	6) 000

Figure 6.   Case 2 profile results.

In Figure 6 the profile results for the *oracle* and the *test program* are shown. The error in case 2 was introduced immediately after the first predicate. The first set of input values produced an error at the third bit. The fourth set of input values produced an error at the second bit and the third bit. Since this simple program flows in a sequential order, it can be deduced that the fault occurred either in the second predicate or somewhere prior to it. It is impossible to rule out that the fault may not exist in predicate one, because the program cannot be exhaustively tested.

The assumptions that have been made can be considered naïve, because it only assumes that there is one fault in the program, which is not the case in the real world. In the preliminary work only one fault was place in the *test program.* The goal was to monitor this simple program to see what the results would be when inserting only one error of commission into the program.

Unlike in case 1, in case 2, one of the sets produced and output where two of the bits were incorrect. An attempt can be made to draw a conclusion from this observation and say that the fault was introduced at the beginning of the program causing errors to be manifested at bit 2 and 3. In case 2 it can be deduced that a fault exist somewhere in the program prior to the second predicate. Another conclusion that was drawn here is that all of the code prior to the predicate must be considered when attempting to isolate faults.

Test program

```
int x1 = 0;      int x2 =0;      int  x3 = 0;
int y1 = 0;     int y2 = 0;      int y3 = 0;
int z = 0;

if (x1 ≤ x2)    then   y1 = x3; y2 = x2;
                else   y1 = x2; y2 = x1;

if (y2 ≤ x3)    then   y3 = x3;
                else   y3 = y2; y2 = x3;

if (y1 ≤ y2)    then   z = y2;
                else   z = y1;

print z;
```

Figure 7. Case 3 Test program.

Figure 7 shows the test program used in case 3 with the error of commission displayed in bold letters. In this case the error was introduced directly after the first predicate.

Correct predicate profiles results	In-correct predicate profiles results
1) 11**1**	1) 11**0**
2) 011	2) 011
3) 00**1**	3) 00**0**
4) 101	4) 110
5) 10**0**	5) 10**1**
6) 0**00**	6) 0**11**

Figure 8. Case 3, error and observations.

The results of the predicate profiles in case 3 look very similar to the results in case 2. The error was introduced in the same location in both cases 2 and 3. The first bit in the incorrect profile are exactly the same as the first bit in the correct profiles in case 2 and 3, this was expected since no errors were introduced in the first conditional statement in both cases.

From a decision making perspective no thought was given at this point to introduce an error in the predicate itself. All of the effort at this point was focused on analyzing errors of commission in all parts of the program except the predicates.

B. PRELIMINARY EXPERIMENTATION

Working with the hand written program initially to get a feel for what statistical debugging looks like quickly lost its luster because of the simplicity of the program. It was time to write some small programs that could be run on a computer so that the programs could be run numerous times. In Figure 8, the results from case 3 are shown; the same conclusion was drawn here as in case 2.

1. First Computer Experiment "Selection Sort Program"

The first program run on the computer was designed to take integers as input and place them in an array. It would then go through the array and sort the numbers in ascending order and find the median. The sorted numbers and the median was output to the screen to review for correctness. The number of times a predicate was visited and the number of times a predicate was true was also output to the screen. A selection sort program was chosen because it had more conditional statements than the other sorting algorithms that were available. At this point in the preliminary experimentation the goal was to find a program that had no more than five to ten predicates since all of the instrumentation of the *target program* was being accomplished manually. Before starting to manually instrument the program a thorough knowledge had to be gained as to what the *target program* does. At this point, the architecture for instrumentation was not in place so; the code that was being tested and the oracle were placed inside the main class. Only one class was being utilized for experimentation. It was not clear if it would be wiser to used one class and place all the code inside or if it was better to utilize multiple classes, only time would tell. From this point on the selection sort program will be referred to as *sortTestx,* this will refer to all the iterations of the original program that was developed.

One of the first orders of business during manual instrumentation was to check the results of the *sortTest1* program for the correctness of result and making sure that all the numbers that were provide by the user were returned in ascending order, sortTest1 is the first iteration of the selection sort program. This checking for correctness was performed by manual inspection. Once satisfied that the program was performing correctly it was time to utilize a Random Generator in Java to populate the array with numbers. This was rather simple to do since the Random Generator is a standard feature in Java and only an instance of the class needed to be instantiated in the program.

Figure 9 and Figure 10 shows the selection sort program that was utilized before any manual instrumentation was performed on it.

```
class Selection
{
public static void main(String arg[])
{
int x[]; //declaring an empty array,initialized later
int i;

//checking if arguments have been given or not at the command
//line
if(arg.length==0)
System.out.println("No  Arguments  Specified...\n");//prints  msg
//if no args specified

else
{
x=new int[arg.lenght]; //initializing the array with the no. of
                       //args specified

try //trapping any errors that might occur
{
for(i=0;i<x.length;i++) //loop for putting the values into the
//array
x[i]=Integer.parseInt(arg[i]);

selectionSort(x); //calling the method for sorting,defined later
                  //in the class

//printing the sorted list
System.out.println("Sorted List:-");

  for(i=0;i<x.length;i++)
     System.out.println(x[i]);
}
catch(NumberFormatException  E)  //traps  the  error  if  specified
                                 //arg is not a number
{
System.out.println("Given argument is not a number...\n");
   }
 }
}
static  void  selectionSort(int  x[])  //method  for  sorting  the
                           //list,made static for using in main
{
int i,j;
int pos=0; //variable to fix a position in array
int min=0; //variable to fix a number as minimum.

for(i=0;i<x.length-1;i++)
{
min=x[i+1]; //setting the minimum number
pos=i+1; //fixing the position to the index of that number
```

Figure 9. First portion of the original Sort class.

```
}
}

if(x[pos]<x[i]) //if number at x[pos] is less than number at x[i]
                //then swap the values
{
x[pos]=x[pos]+x[i];
x[i]=x[pos]-x[i];
x[pos]=x[pos]-x[i];
       }
     }
   }
}
```

Figure 10. Second portion of the original Sort class.

As previously stated the first thing that had to be accomplished was to test and see if the selection sort program worked properly. Of course it is impossible to do exhaustive testing but some basic testing was performed to see if the program would meet the criteria. Four different simple tests were performed on the selection sort program. First a test was performed using negative integers only. Second a test was performed using positive integers only. Third a test was performed using positive and negative integers and last a test was performed using input other than integers. In all four tests the program worked as expected. The rationale behind doing these basic test was to test all possibilities that where of the upmost concern. The only requirements that the selection sort program had to meet, was to be able to sort all possible combinations of positive and negative integers.

The next order of business was to write a class that accumulates all the profile information during the run of a *test program* after it has been manually instrumented. The name of the accumulator class that was written to perform this task is the *Counter3*. The *Counter3* class has six integer arrays, one constructor and two methods. The initial decision was to count all true and false responses for all instrumented predicate. As time went by it was realized that the total numbers of time a predicate was visited encompasses both true and false visits, with that said only true predicates responses needed to be recorded. The difference between the total number of visits and total number of true predicates results in the total number of false predicate visits. Since the decision was made at this point to place the *test program* and the *oracle* inside the main class, it was assumed that only one predicate method was needed inside the *Counter3 class* and would be able to monitor predicates in the *test program* and *oracle program*. Figure 11 and Figure 12 show the code of the *Counter3* class which is used to accumulate predicate profile information.

```
package thesis_work;

public  class Counter3 {

     int numberofVisit [];
     int numbertrue [];
     int numberofVisitCorr [];
     int numbertrueCorr [];
     int numberofVisitInCorr [];
     int numbertrueInCorr [];

    //Constructor
     public Counter3(int num) {
        numberofVisit =  new int [num+ 1];
        numbertrue = new int[num + 1];
        numberofVisitCorr =  new int [num + 1];
        numbertrueCorr = new int [num + 1];
        numberofVisitInCorr = new int [num + 1];
        numbertrueInCorr = new int [num + 1];

        for (int i =0; i <= num; i++){
            numberofVisit  [i] =0;
            numbertrue [i]= 0;
            numberofVisitCorr [i]= 0;
            numbertrueCorr [i]= 0;
            numberofVisitInCorr [i]= 0;
            numbertrueInCorr [i]= 0;
        }
    }
      /*this  predicate  method  monitors  the  results  of
instrumented
       predicates*/
     public  boolean predicate (int n, boolean b){

        numberofVisit[n]++ ;

        if (b){
            numbertrue[n]++;
```

Figure 11. First half of the *Counter3* class.

```
}
        return b;
    }

    //print_array methods was used in to verify the predicate
    //method was functioning correctly during the preliminary
    //experimentation.
    public void print_array(int num){

        System.out.println("This is!!!!");
        for (int i =0; i <= num; i++)
            System.out.println("\t total  " +  numberofVisit[i]  +
                                  " \t numbertrue  " + numbertrue[i]);
    }
}
```

Figure 12. Second half of “Counter3” class.

The code in the *Counter3* class reference as (Figure 11 and 12) is highlighted in red because this is code that did not previously exist; it had to be written for the purpose of monitoring the predicates in the selection sort program.

Now it was time to manually instrument the selection sort program, this portion of the preliminary experimentation was very time consuming and difficult. One reason was because a methodology for instrumenting the *target program* was understood in theory, but the challenge in real life is that all programs are different and building a test driver for a program has to be tailored to the needs of the specific *target program*. Many mistake where made in attempting to instrument the selection sort program and the majority were due to the fact that an intimate knowledge must be obtained about what all the code in a target program does must be obtained.

The first mistake was the attempt to have two different types of arrays for storing predicate profile information. Local arrays would be in the main class and accumulator (global) arrays would be used to store the results of the difference between local arrays, they are in the Counter3 class. The idea was to have the *oracle* and the *test program* both located inside the main program. Local arrays would be used to store the results of each predicate during a single run and the difference between the *oracle* and *test program* predicates would be recorded in the arrays in the *Counter* class. After several iterations of the selection sort program, a successful implementation of this methodology was performed on the Triangle program. Before the Triangle program is discussed, code of the instrumented sortTes4 program will be displayed and some of the pitfalls and difficulties that were encountered will be discussed. The *sortTest4* program is being displayed because it was the fourth attempt at making the selection sort program work properly and it comprises all the mistakes that were made. All of the instrumentation code will be highlighted in red and the original program will be in black text.

Figure 13 thru Figure 16 display the code of the sortTest4 program. The selection program uses a method call within the program to call the *selectionSort* method that actually sorts the numbers that are provided. Starting at the beginning of the SortTest4 program there is the standard format used in java, the package name is thesis_work. The class name is SortTest 4, after the class name an instance of the *Counter4* class was declared as static, since only one instance needed to be instantiated. An obvious pitfall can be seen, it was not clear at this point if the static Counter class should be used or the "Counter4 global2 = new Counter4 (5)" should be used. The decision was made to go with the static approach because it would be possible to inadvertently generate many instances of *Counter4* and that was not the intent. What has needed was only one instance of the *Counter4* class that could be used throughout the execution of one run of the program.

Another pitfall encountered was placing the declaration for the instance of the *Counter4* class inside of the *selectionSort* method. This problem was not clear initially, but once some thought was given to understanding what the program was doing it was clear that every time the selectionSort method was called a new instance of the Counter4 class was instantiated which is not a good thing. Another pitfall that was encounter was being unsure as to what exactly need to be monitored. In the case of this selection sort program, the dilemma was, should an emphasis have been placed on only trying to instrument the selectionSort method or the entire program? Ultimately it was decided that the focus needed to be placed on instrumenting the entire program because the objective is to find faults. Some programs have code that may seem unimportant in order to execute the program. It is important to remember that debugging entails looking at the entire program. With that said the decision was made to find another program and to start off with a fresh beginning.

```
package thesis_work;

import java.util.*;
import java.util.Arrays;

public class SortTest4 {

    static Counter4 global1 = null;

    public static void main(String arg[])
    {
        global1 = new Counter4(5);
        Random generator = new Random();
        Counter4 global2 = new Counter4(5);

        final int ARRAY_SIZE = 100;

        int i;
        int b;

        int x[];
        int y[] = null;

        int countNumbers;

System.out.print("This is just a test program for my thesis work,
\n" +   "my goal is to take in numbers from a user. Return the\n"
+  "numbers in ascending order and print out the median.\n" +
  "I will count the number of times the program iterates\n" +
  "through each looping statement as well!\n\n");

        //checking if arguments have been given or not at the
        //command line

        for(int k=0; k<1000; k++) {

            if (arg.length==0){

                x = new int[ARRAY_SIZE];
                y = new int[ARRAY_SIZE];

                for(int j=0; j<ARRAY_SIZE; j++){

                    x[j] = generator.nextInt(10000);

          //populating array y with the same numbers as array x.
                    y[j] = x[j];

                }
```

Figure 13. sortTest4 class.

```
  }

                //prints message if no arguments are specified
                //System.out.println("No Arguments Specified from
command line Random " +
                //                   "generator utilized...\n");

            }else
            {
                x = new int[arg.length]; //declaring an empty
array,initialized later

                for(i = 0; i< x.length; i++) //loop for putting the
values into the array
                    x[i]=Integer.parseInt(arg[i]);
            }

            countNumbers = x.length;

            System.out.print("You entered " + countNumbers + "
numbers to be " +
                    "sorted." + "\n\n");

            try //trapping any errors that might occur
            {

                selectionSort(x); //calling the method for
sorting,defined later in the class

                //printing the sorted list
                System.out.println("\n\nHere is the sorted list:-");

            }

            catch(NumberFormatException E) //traps the error if
specified arg is not a number
            {
                System.out.println("The given argument/s is not a
number...\n");
            }

            System.out.println("\n");

            int middle = x.length/2;  //subcript of middle element
            if (x.length%2 == 1) {

                //Odd number of elements -- return the middle one.
                System.out.println ("Median is " + x[middle] +
"\n");
```

Figure 14. sortTest4 class continued.

```
}else
            {
                // Even number -- return average of middle two
                // Must cast the numbers to double before dividing.
                System.out.println ("Median is " + (x[middle-1] +
x[middle]) / 2 + "\n");

            }

            Arrays.sort(y);

            for(i=0; i< x.length -1; i++){


                System.out.println(x[i] +" " + y[i]+ "\n");

                boolean arr_equals = Arrays.equals(x, y);
                if (arr_equals == true){

                    //true counter incremented
                    global1.predicate(5, 3>1 );

                }else{

                    //false counter incremented
                    global1.predicate(5, 3<1 );
                }
            }
        }
    }

    private static void sort(int[] y) {
        // TODO Auto-generated method stub

    }

    static private void selectionSort(int x[]) //method for sorting the
list,made static for use in main
    {

        //Counter4 global1 = new Counter4(4);

        int i,j;
        int pos=0; //variable to fix a position in array
        int min=0; //variable to fix a number as minimum.

        for(i=0; global1.predicate(1, i >x.length-1); i++)
        {
```

Figure 15. sortTest4 class continued.

```
min=x[i+1]; //setting the minimum number
pos=i+1; //fixing the position to the index of that number

            for(j=i+1; global1.predicate(2,j<x.length); j++)

            {

                if(global1.predicate(3,min > x[j])) //if number at
//x[j] is greater than minimum then


                {

                  min=x[j]; //put x[j] in min
                  pos=j; //and change position to index number of j

                }
            }

            if(global1.predicate(4, x[pos]<x[i])) //if number at
//x[pos] is less than number at x[i] then swap the values

            {

                x[pos]=x[pos]+x[i];
                x[i]=x[pos]-x[i];
                x[pos]=x[pos]-x[i];
            }
        }
        global1.print_array(5);
        global1.deltas(5);
    }
}
```

Figure 16. sortTest4 class continued.

2. Second Experiment "Triangle Program"

After spending a few weeks on the selection sort program, the decision was made to find a new program to experiment on, using the information that had been gained from working on the selection sort program to move forward. Some pseudo code was available that represented the logic of a Graphical User Interface tool that could take in 3 integers and return if they could be utilized to comprise a triangle. If the integers could be used to form a triangle the tool would return to the user what type of triangle they would make i.e., Right triangle, Isosceles Triangle or an Equilateral Triangle. The pseudo code that was available was taken and the program was written in java so that it could be used for experimentation purposes. The pseudo code was written very well so it was easy to convert into java code.

Integers were randomly generated and placed into an array. Two copies of the integers are stored in different variables so that they can be feed into the *oracle* and *test program* respectively. The decision was made during manual instrumentation that all arrays used in monitoring the profile of predicates will be located in the Counter class. It did not matter if the arrays were used to store each individual run of a program or if the array would store information about multiple runs, it would still be located in the Counter class.

Upgrades where made to the Counter3 class that were used in the selection sort program, two methods were added to assist in monitoring predicate profiles. The method *compare* was the first method that was added to the Counter class. The *compare* method was responsible for comparing the results of the *oracle* and *test program* profile for one single run. The *increment* method was the second one added. It was responsible for accumulating the results from comparing the *oracle* and *test program* for a single run. Figures 21-23 display the upgraded *Counter3* class, the name of the class was changed to *Counter4* to represent the next iteration of this class.

```
package thesis_work;

/**
 * @author Toriano Murphy
 *
 */
public class Counter4 {

      private int numberofVisit[];
      private int numbertrue[];
      private int numbernottrue[];
      private int numberofVisitCorr[];
      private int numbertrueCorr[];
      private int numberofVisitInCorr[];
      private int numbertrueInCorr[];

      public Counter4(int num) {
            numberofVisit = new int[num + 1];
            numbertrue = new int[num + 1];
            numbernottrue = new int[num + 1];
            numberofVisitCorr = new int[num + 1];
            numbertrueCorr = new int[num + 1];
            numberofVisitInCorr = new int[num + 1];
            numbertrueInCorr = new int[num + 1];

            for (int i = 0; i <= num; i++) {

                  numberofVisit[i] = 0;
                  numbertrue[i] = 0;
                  numbernottrue[i] = 0;
                  numberofVisitCorr[i] = 0;
                  numbertrueCorr[i] = 0;
                  numberofVisitInCorr[i] = 0;
                  numbertrueInCorr[i] = 0;

            }
      }
      public boolean predicate(int n, boolean b) {

            numberofVisit[n]++;


            if (b) {
                  numbertrue[n]++;

            } else {
                  numbernottrue[n]++;

            }
            return b;
      }
```

Figure 17. Counter4 class used with Triangle.

```
}

    public void compare (int n){

        for (int i = 0; i < 6; i++){


            if(numbertrue[i] == numbertrue [i + 6] ){

                this.increment(i, 1==1);

            }else{
                this.increment(i+6, 1==0);

        }
    }
        for (int i = 0; i < 6; i++){
             numbertrue[i] = 0;
             numbernottrue[i] =0;

        }

   }


    public boolean increment (int i, boolean c){

        numberofVisit[i]++;


        if (c){

            numbertrueCorr[i]++;

        }else {

            numbertrueInCorr[i]++;


        }


        return c;
    }
```

Figure 18. Counter4 class used with Triangle.

```
}

      public void compare(int n) {

            for (int i = 0; i < 4; i++) {


                  if (numbertrue[i + 4] == numbertrue[i + 8]) {


                        this.increment(i + 4, 1 == 1);

                  } else {
                        this.increment(i + 8, 1 == 0);

                  }
            }
            for (int i = 0; i < 5; i++) {
                  numbertrue[i] = 0;

            }

      }

      public boolean increment(int i, boolean c) {

            numberofVisit[i]++;


            if (c) {

                  numbertrueCorr[i]++;

            } else {

                  numbertrueInCorr[i]++;

            }

                                    return c;
      }
```

Figure 19.   Counter4 class continued.

```
public void deltas(int num) {

            for (int i = 0; i < num; i++) {

                  System.out
                              .println("Delta "
                                          + i
                                          + "\t\t"
                                          + (Math
                                                      .abs((((float)
numbertrueCorr[i] / (float) numberofVisit[i]))
                                                                  -
(float) numbertrueInCorr[i]
                                                                  /
(float) numberofVisit[i] - 1)));

            }

      }

      public void print_array(int num) {

            System.out.println("Here is the statistical data!!!!\n\n");
            for (int i = 0; i <= num; i++) {

            }

      }

}
```

Figure 20. Counter4 class continued.

The source code to the *Triangle* Program can be found in the appendix section of this thesis. The manual instrumentation process of the *Triangle* program did take some time but things went faster here because the basic idea about what needed to be done was understood after working with the selection sort program. Once the *test program* of the Triangle program was instrumented a copy of this program without any faults introduced was used as the *oracle*.

During the instrumentation of the Triangle program it was realized that automatically instrumenting the test driver would be impossible, because all programs have different needs in order to run. The goal for the test driver would be to make it as generic as possible.

The purpose of the Triangle program was to take in three integers and return what type of triangle those integers produced. If the integers did not produce a triangle, the program would let the user know.

C. PRELIMINARY DATA AND RESULTS

Predicate	visits	true visits	% true visits	delta
P1	1	0	0	**0.999999**
P2	0	0	#DIV/0!	#DIV/0!
P3	0	0	#DIV/0!	#DIV/0!
P4	0	0	#DIV/0!	#DIV/0!
P5	0	0	#DIV/0!	#DIV/0!
P6	0	0	#DIV/0!	#DIV/0!
P7	0	0	#DIV/0!	#DIV/0!

Figure 21. Error in predicate 1.

Figure 21 shows the results when an error was introduced in predicate one of the triangle program. Since predicate one is a for-loop and an error was introduced directly in the predicate, the following predicates 2–7 were not visited resulting in a division by zero error. Predicate 1 has a delta of almost 1; a delta of this magnitude is the strongest indication possible that an error exist in the test program.

Predicate	number of visits	number true visits	% true visits	delta
P1	1000001	1000000	0.999999	0
P2	1000000	0	0	**0.75**
P3	0	0	#DIV/0!	#DIV/0!
P4	0	0	#DIV/0!	#DIV/0!
P5	0	0	#DIV/0!	#DIV/0!
P6	0	0	#DIV/0!	#DIV/0!
P7	0	0	#DIV/0!	#DIV/0!

Figure 22. Error in predicate 2.

Figure 22 shows the result of introducing an error directly into predicate 2. It has the strongest indication out of the seven predicated being monitored, in this case give a true-positive result. It is interesting to note that predicate 1 in this case has a delta of zero. This means the predicate 1 profile is identical in the *oracle* and *test program*.

Predicate	number of visits	number true visits	% true visits	delta
P1	1000001	1000000	0.999999	0
P2	4000000	3000000	0.75	0
P3	0	0	#DIV/0!	#DIV/0!
P4	1000000	0	0	**0.323353**
P5	0	0	#DIV/0!	#DIV/0!
P6	0	0	#DIV/0!	#DIV/0!
P7	0	0	#DIV/0!	#DIV/0!

Figure 23. Error in predicate 4.

Figure 23 shows the result of introducing an error directly into predicate 4. It has the strongest indication out of the seven predicated being monitored, in this case giving a true-positive result.

Predicate	number of visits	number true visits	% true visits	delta
P1	1000001	1000000	0.999999	0
P2	4000000	3000000	0.75	0
P3	0	0	#DIV/0!	#DIV/0!
P4	1000000	324411	0.324411	0.001058
P5	324411	61308	0.188982494	**0.134651778**
P6	263103	140443	0.53379475	0.066283737
P7	122660	17765	0.144831241	0.002726408

Figure 24. Error in predicate 5.

Figure 24 shows the result of introducing an error directly into predicate 5. It has the strongest indication out of the seven predicated being monitored, in this case giving a true-positive result. In this case predicate 6 has a relatively strong indication that an error exist as well, even though the error of commission is not in this predicate. This is the case because the incorrect results from predicate 5 are used in predicate 6, casing a delta of a significant size.

Predicate	number of visits	number true visits	% true visits	delta
P1	1000001	1000000	0.999999	0
P2	4000000	3000000	0.75	0
P3	0	0	#DIV/0!	#DIV/0!
P4	1000000	323630	0.32363	0.000277
P5	323630	17657	0.054559219	0.000228503
P6	305973	244663	0.799622843	**0.199544356**
P7	61310	8739	0.142537922	0.000433089

Figure 25. Error in predicate 6.

Figure 25 shows the result of introducing an error directly into predicate 6. It has the strongest indication out of the seven predicated being monitored, in this case giving a true-positive result.

Predicate	number of visits	number true visits	% true visits	delta
P1	1000001	1000000	0.999999	0
P2	4000000	3000000	0.75	0
P3	0	0	#DIV/0!	#DIV/0!
P4	1000000	323418	0.323418	6.5E-05
P5	323418	17490	0.054078623	0.000252093
P6	305928	183242	0.598971	0.001107487
P7	122686	11712	0.095463215	**0.046641618**

Figure 26. Error in predicate 7.

Figure 26 shows the result of introducing an error directly into predicate 7. It has the strongest indication out of the seven predicated being monitored, in this case give a true-positive result.

Figure 21 through 26 shows results from the *Triangle* program runs. In all the case, extensive testing was performed where the fault was introduced directly into the predicate. Since the input data could be randomly generated, the decision at this point was to analyze the ability to point to the fault in these cases. In these cases where the fault was introduced directly in the predicate, this technique was able to point to the predicate that had the fault 100% of the time in the test that were ran, this does not represent all possible cases. Of course the code inside a predicate only makes up a small portion of a program and therefore additional testing must be performed where faults are introduced into other parts of the program, like assignment statements or code that does calculations.

The data produce by the *Triangle* program seemed promising, at least when introducing one fault in the program. On that note, it seemed as though statistical debugging at least in the context of this thesis work may be promising when looking at faults that reside inside the predicate. It may be limited in the ability to find faults in a program outside of the predicate, because the predicate itself is the indicator that acknowledges that a fault may be in the program. The best case scenario is that the fault lies in the predicate itself; thus far in the Triangle program experiment this technique has a strong tendency to isolate the fault. Up to this point, the experimentation and instrumentation process of a program was very time consuming. This needed to be automated in order to run more experimentation on different types of programs a lot faster and more efficiently.

Another challenge that presented itself is how to obtain results from a program. The problem is that in the real world programs do more than just sort numbers or check to see if three numbers make a triangle. Therefore, the challenge of obtaining the result of a program is important. Once results have been obtained it is easy to check them for correctness.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. DESIGN OF INSTRUMENTATION TOOL

During the preliminary experiments, the amount of work required to manually instrument a program became apparent. More important, by designing a tool to automatically instrument a target program, the introduction of error into the instrumented program is reduced and the speed at which a program could be instrumented for testing is increased. It is more practical to use an instrumentation tool because of the size of programs today. Trying to manually instrument a program with several thousands of lines of code would be time consuming. Testing and debugging is very expensive and any time a tool can be used to minimize the time needed in instrumenting a program, it must be welcomed.

There are two steps that must be accomplished before the proposed method of statistical debugging utilized in this thesis research can be accomplished. First, a *target program* must be instrumented with the code that will perform the monitoring of the target program during runtime. Second the program must be manually set up in a test suite. As of today, there is no tool available that will "fully" instrument all types of program, the nature of the program must be understood first, i.e., what is the program doing, what type of input is required to run the program and how are the results obtained and checked for correctness. In this thesis a tool was designed to do the instrumentation of a *target program* predicates automatically and a methodology was constructed for building the test driver using a generic template, the test driver aspect is still instrumented manually. An effort was made to derive a generic solution for the test driver. The needs of the *target program* will dictate who much manual instrumentation will be necessary to perform on the generic test driver.

The instrumentation aspect can be broken down into three steps. First predicates have to be instrumented and this task is performed automatically using the instrumentation tool. The predicates in the oracle and the test program are instrumented automatically. Second the instrumented test run is manually instrumented, because the needs depend on the program. Third the instrumented oracle run is manually instrumented because it has the same needs as the test run.

A. ARCHITECTURE OF INSTRUMENTATION TOOL

The architecture of this instrumentation tool can be viewed as Aspect Oriented Architecture. Several classes work together cross-cutting to produce one executable entity. This section will discuss and elaborate on the first job of the instrumentation tool. Next the entire test suite used in conducting the statistical debugging will be viewed and discussed.

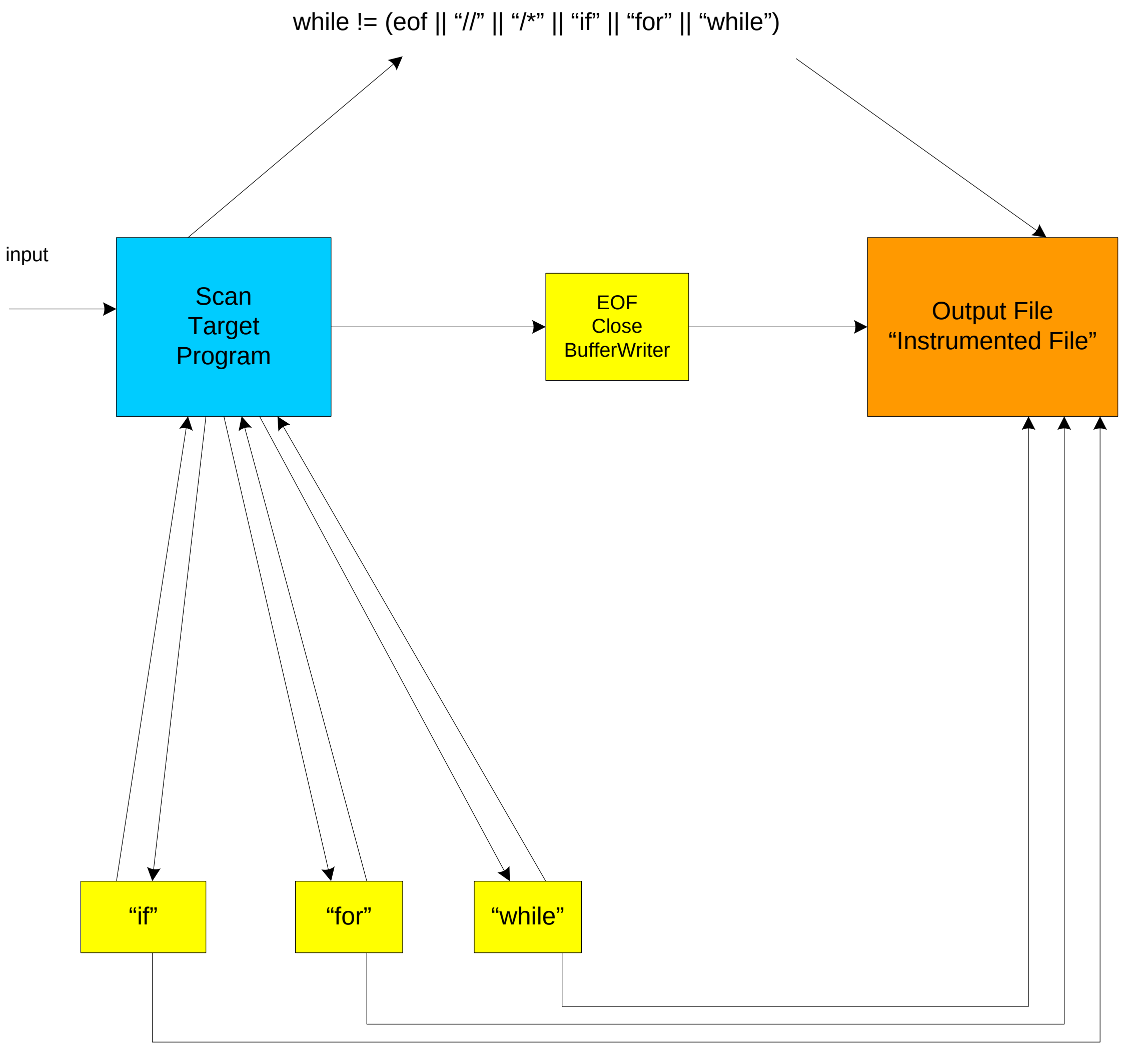
while != (eof || "//" || "/*" || "if" || "for" || "while")
input
Scan Target Program
EOF Close BufferWriter
Output File "Instrumented File"
"if"
"for"
"while"

Figure 27. Instrumentation tool diagram.

In Figure 27, a display of the instrumentation tool shows a flow diagram of how the instrumentation code is inserted into a *target program.* The instrumentation tool takes in two arguments an input file (*inFile*) which is the target program ready to receive the instrumentation code. The second argument is the output file (*outFile*), this file is the instrumented program which two copies will be used in the experimentation and will be covered more in detail in the next chapter. The instrumentation tool reads in the *target program* one character at a time. The BufferedReader and BufferedWriter class in java allows "characters" to be read in and to be written to a file. While the *target program* is being read in, the instrumentation tool will continue to read the *target program* in and write it to the output file. When the instrumentation tool encounters the keywords, "end of file, "//", "/*", "if', "for" or "while" the corresponding methods that handle the instrumentation of these keywords will take over. In the case of "//" the instrumentation tool understands that this is a comment line and will just copy everything it sees into the output file until the end of line is reached. Once the end of line is reached the program continues to scan the file until the while condition is false. In the case of "/*" the program will copy everything into the output file until a "*/" is seen. It does not matter if an "if', "for" or "while" keyword is encountered in a comment section, it is just ignored. In the case of the key word "if, "for" and "while" are encounter outside a comment line, the associate method for each is called. The "if statement' and "while statement" are instrumented in the same manner.

```
sortTest1:

1.  for (i=0; global.predicate (1, i<x.length-1); i++)

2.  if (global.predicate (3,min>x[j] ) )
```

Figure 28. Examples of instrumented predicates.

In Figure 28, the "if statement" prior to instrumentation was **if (min>x[j]).** The key to instrumentation is to look for the first open parenthesis after the "if statement" an insert **global.predicate(x,** which is highlighted in red. A counter is used to keep track of the number that will be inserted after the second parenthesis. In the case of a "while statement", this instrumentation is performed exactly the same as the instrumentation for an "if statement".

The solution to the for-statement had to be approached in a different way. The key in instrumenting the "for statements" was to find the first semicolon after the keyword. In Figure 28 immediately after the first semicolon the instrumentation code appears in read. Next, prior to the second semicolon a close parenthesis was inserted in the statement. This instrumented code is written to the output file. Once the instrumentation of the "for" statement is completed the instrumentation program continue to scan and instrument the target program until the end of file is reached. Once the end of file is reached, the BufferedReader and the BufferedWriter are both closed.

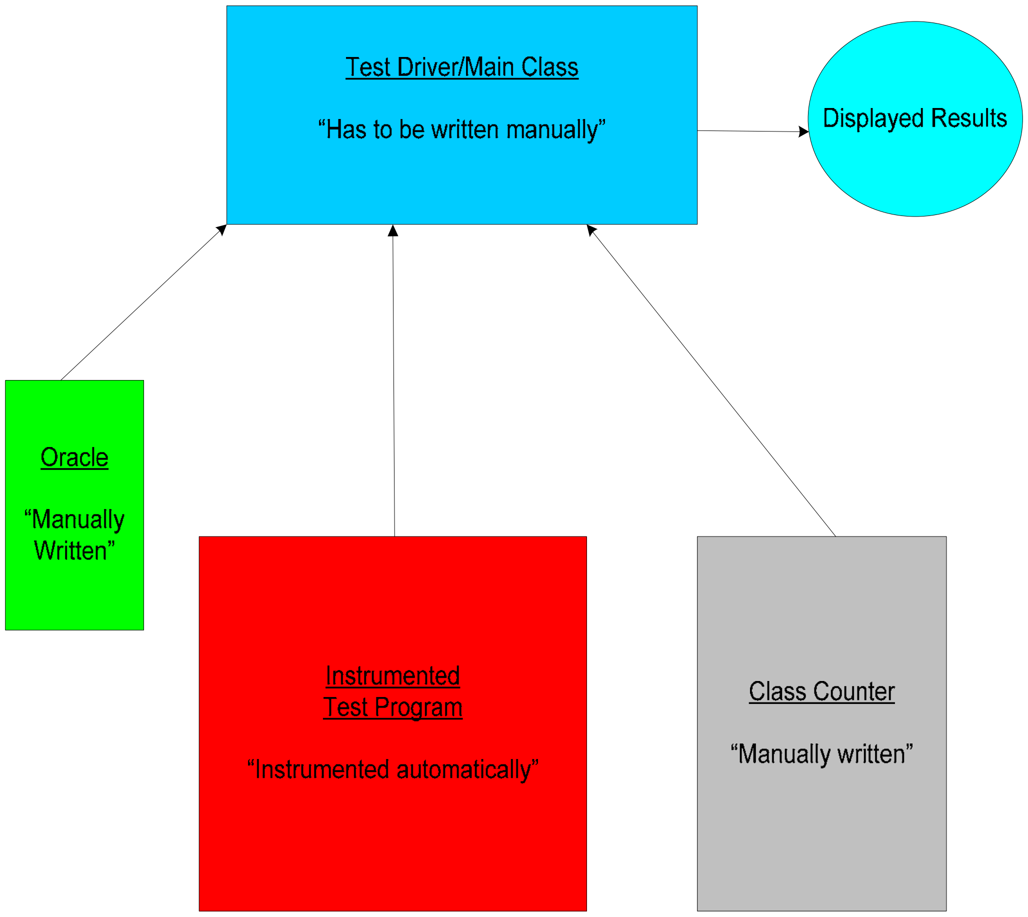

Figure 29. Instrumentation Test Suite.

Figure 29 shows the instrumentation test suite as a whole. The test driver is where the main program is located; this class is responsible for calling the other classes. The *Instrumented Target Program* is the *target program* that has received all the required instrumentation code; it is adjusted as need to fit properly in the test driver.

The *Oracle* class is a copy of the Instrumented Target Program, this class will be used to verify and check the results of the *test program* during runtime. Errors of commission are inserted into the *test program*. The goal is to see if faults in a program can be identified by monitoring the behavior of predicates during the run of a program, using an *oracle* to confirm when the results of a predicate are correct or incorrect.

The *Counter Class* performs all the monitoring during the execution of a test. This class is where the *predicate* method is called by the main program. The data structure (arrays) that are utilized to collect the predicate results is located here. The method which calculates the difference in predicates in the program that is being tested and the *oracle* is located in the *Counter Class* as well.

There are four methods inside the *Counter Class* that perform the bulk of the work. The *sumvisistcorrect* method is called by the *oracle* to calculate the number of times each predicate was visited and each time they visited predicate was true. The predicate method is responsible for incrementing the array count at the array position identified by the first argument that is taken in by the predicate, which is a number. The predicate method will automatically increment the *numberofVisitA* array in the Counter Class. In the cases where the predicate is true the *numberTrueA* array will be incremented. This *predicate* method is called by the *oracle class*. The *predicateB* method is called by the *test program* and performs the same duty as the *predicate* method in the *oracle* class. The *numberofVisitB* array and the *numberTrueB* array are incremented by the *predicateB* method. The delta method is called once the main loop in the main program is complete. The delta method calculates the absolute difference between the predicates in the *Oracle* that are associated with the same predicates in the *test program*.

B. PROBLEMS ENCOUNTERED WHILE DESIGNING THE TOOL

The initial challenge was trying to decide on which programming language should be utilized to code the instrumentation tool. The decision in the beginning was to use Rudy, based on the fact that it is object-oriented, there is no need to declare variables before they are used and it uses minimal punctuation just to mention a few things great about the language. The decision in the end was to use java on the bases that the majority of the programming experience was in java and java is a mainstream language. The instrumentation tool could have been written in Rudy more efficiently or even in Python which is another power programming language

1. Conversion

The majority of the challenges that where encountered were minor. One problem that took a little research to find a solution having integers defined in the instrumentation program and the integers needed to be written to the output file as a character, this seemed simple at first. Since the BufferedWriter and BufferedReader classes were being used everything was being manipulated as character. When the time came to insert instrumentation code, the BufferedWriter would only write out characters as it was designed to do.

```
Conversion Problem

bw.write(String.valueOf(Counter) );
```

Figure 30. Conversion Problem.

Figure 30 shows the code that was used to address the conversion problem; bw.write(char) only takes characters as an argument. By inserting the code in red as an argument it was possible to convert *counter* which is an integer into a character. This code is not a part of the BufferedWriter class in Java, but the nature of the problem was understood and with a little searching a solution was found.

2. Global Counter

The global variable as understood in a traditional since is not allowed in Java. A variable that is created with-in a class is only viewable by that class. The dilemma that existed in this research was that a global variable was needed to hold the predicate results while the *Oracle* and *Test Program* were run up to one hundred thousand times during a single test. This problem was rectified by using what is known as a Singleton Class in Java. The singleton design pattern allows only one instantiation of a class. This can be useful when trying to coordinate actions across multiple classes, as was the problem in this thesis research. Although the singleton pattern is not called a global pattern, it has all the characteristic and behaviors of a global entity.

```
From main class:

static Counter5 global = Counter5.getcounter5Object();

From Counter class:

public static synchronized Counter5 getcounter5Object() {

     if (counter5Object == null){
     //it's ok we can call this constructor
      counter5Object = new Counter5(15);
     }
     return counter5Object;

      }

//  prevent the creation of another instance of this Counter5
class
    public Object  clone()

     throws CloneNotSupportedException
     {
     throw new CloneNotSupportedException();

     }
    private static Counter5 counter5Object;
```

Figure 31. Solution to global class problem.

In Figure 31, the code that solves the problem of having a global class is shown. The method used to create the one instance of the *Counter5* class along with the code to ensure only one instance of the class is instantiated is here. The code used in the main class is the exact code used in the *Oracle* and the *test program* class. The beauty of the singleton is that the one instantiation of the *Counter5* class is coordinated across all the classes by the test driver/main program.

V. BIG EXPERIMENT

There are few things that must be addressed prior to talking about the main experimentation. For the sake of clarity the term *target program* refers to a program that has been identified as a candidate program to conduct experimentation on. Once the target program has gone through the automatic instrumentation tool and has been manually instrumented as needed to run in the test driver, it will be referred to from there on a the *test class*. The *test class* and the *oracle class* are identical program. During experimentation errors of commission will be placed inside the *test class* so that the results can be analyzed. In order to perform the experiments, programs where the input could be randomly generated are the only programs that could be utilized for this research. This made the search for *target programs* a challenge. Finding programs that met the criteria of having 10 or more predicates and having input that could be randomly generated became a challenge. Hundreds of programs were found that were performed complex mathematical computations. The problem with these programs is that the methods in the programs had random generators inside. By inventing an instrumentation tool, the tool itself is limited in the assistance that it offers and can only perform the task it has been programmed to do. Trying to use programs with multiple random generators would b to much to deal with at this time.

The first step before any experimentation was started was to run the program several times to see if it worked. Along with seeing if the *target program* worked, a basic knowledge as to what the *target program* did had to be understood.

The target programs where taken from www.java2s.com [8]. This site is essentially a repository of source code free to the public. The source code is allocated according to topic making it easy to find of interest.

A. EXPERIMENT 1

The program utilized in this experiment was written to search for the minimum of a multivariable function through a steepest-descent method.

First, the *target program* was run through the instrumentation tool. The instrumentation tool takes in two arguments an input file and an output file. The *target program* was copied and pasted into the input file. Next the instrumentation tool was run, after running the *target program* through the instrumentation tool the output file had to be manually instrumented in order to run in the test driver. All the code that was of no use was manually removed from the *target program*.

```
public static void main(String argv[]) {

        MinimumA minA = new MinimumA(null, 0, 0);
        MinimumB minB = new MinimumB(null, 0, 0);



        for(int k=0; k < 100000; k++){

            double delA = 1e-6, aA = rand.nextDouble();
            double delB = 1e-6, aB = aA;
            double xA[] = new double[2];
            double xB[] = new double[2];

            xA[0] = 0.1;
            xB[0] = 0.1;

            xA[1] =  -1;
            xB[1] =  -1;

            minA.steepestDescent(xA, aA, delA);
            minB.steepestDescent(xB, aB, delB);

      //This  method  keeps  track  of  the  predicate  count  for
      //each run
            global.sumvisitcorrect(6);


        }
        //This  method  calculates  the  absolute  difference
between the same predicates
        //in the oracle and the same predicates in the testing
program.
        global.deltas(6);
    }
```

Figure 32. Experiment 1 test driver/main class.

Figure 32 shows the test driver that was used to run Experiment 1. As you can see an instance of the *Oracle class* (MinimumA) and an instance of the *test class* were created so that they could be run 100,000 times in the main loop. Looking at the variables, it is obvious to see that the same input data was utilized in both classes. A random double is generated and stored in “aA” and “aB” every time the "for" loop executes. The instance (minA) of MinimumA class calls its *steepestDescent* method. Once minA.steepestDescent( ); is finished executing minB.steepestDescent( ); executes. Temporary arrays are used to store the results for iterations through the loop. Before the loop iterates again the results are stored in the accumulator arrays in the *Counter5* class. The delta method from the *Counter5* class is used to calculate the absolute difference between the predicates in the *Oracle class* and the predicates in the *test class*.

```
public static void deltas (int num) {

            for (int i =0; i < num; i++) {

                  System.out.println("Delta " + i + "\t\t" +

(Math.abs((((float)numbertrueCorrA_tot[i]/(float)numberofVisitCorrA_to
t[i]))
                                          -

(float)numbertrueCorrB_tot[i]/(float)numberofVisitCorrB_tot[i] )));

            }
      }
```

Figure 33. Delta method in Counter5 class.

In Figure 33, the math equation that calculates the absolute difference is shown. The result is going to be a number between 1 and 0 in most case, a rare case was encountered but no clear explanation could be derived for this phenomena. In the event no errors of commission have been place in the *test class*, the delta in all cases will be 0.0.

1. Experiment 1 Results

In this subsection the results of several experiments will be presented and evaluated.

```
for (int i=0;global.predicateB(0, i>n);++i) dg += fi[i]*fi[i];
 //Error was introduced in conditional statement.  It should be:
 //for (int i=0;global.predicateB(0, i<n);++i) dg += fi[i]*fi[i];

Delta 0        0.6666667
Delta 1        0.9652778
Delta 2        NaN
Delta 3        NaN
Delta 4        NaN
Delta 5        0.0
```

Figure 34. Experiment 1 test 1.

Figure 34 shows the results of test 1 run. An error of commission "i<n" was introduced into the program inside of predicate zero. In this case predicate 1 is indicating that it has the greatest probability of being the predicate where the faults may exist. Predicate 0 is showing a delta of 70 percent but it is not a strong as a delta of 96 percent. In this research the luxury of knowing exactly where the fault was inserted helps to discern the ability of this statistical debugging technique to find faults with any degree of precision. Even though the guilty predicate does not have the highest percentage, it does have the second highest which is better than having to look through the entire program trying to find the fault with no sense of direction on where to start looking.

```
while (global.predicateB(1, (dg < del)))
//should be:
while (global.predicateB(1, (dg>del)))

Delta 0        0.0
Delta 1        0.96268654
Delta 2        NaN
Delta 3        NaN
Delta 4        NaN
Delta 5        0.0
```

Figure 35. Experiment 1 test 2.

Figure 35 shows the results of test 2 run. In this test, the error of commission was introduced inside of the predicate. The greater than signed was changed to a less than sign. The guilty predicate having the fault has the highest delta, which in this case is 96 percent.

```
for (int i=0;global.predicateB(2, i>n);++i) x[i] -= b*fi[i];
//should be:
//for (int i=0;global.predicateB(2, i<n);++i) x[i] -= b*fi[i];


Delta 0             0.0
Delta 1             0.009482682
Delta 2             0.6666667
Delta 3             0.0
Delta 4             0.46608564
Delta 5             0.0
```

Figure 36. Experiment 1 test 3.

Figure 36 shows the results of test 3; just like in test 2 the guilty predicate in this case has the largest delta. Several other test where done where errors were only placed inside the predicate themselves. In the majority of these test the predicate that had the fault introduced in it had the largest delta. In Experiment 1 the focus was on analyzing the ability of this technique to indicate that a single error was introduced in the predicate. In these three test cases there was l test where the guilty predicate did not have the largest delta, but the delta shown was convincing enough to indicate that a fault may exist there.

In test 3, the guilty predicate that does not present the strongest indication that an error is present. This is important to note because the mechanism being used to indicate the presence of a predicate is the predicate profile which only makes up a small portion of any program. What is promising is that the indication was the second largest indication, so it is still better than no indication at all.

B. EXPERIMENT 2

The same steps for instrumentation in Experiment 2 were done just like the *target program* was instrumented in Experiment 1. In Experiment 2 a program was written for testing purposes to include multiple conditional statements. A method called the *funmethod* was written in this target program that called two other methods, *funmethod2* and *funmethods3*. This program was written with the mind-set of having a *target program* that would have more than 10 predicates. This program has 15 predicates and the results of the test that where run will be discussed here.

1. Experiment 2 Results

```
Delta 0             0.0
Delta 1             0.0
Delta 2             NaN
Delta 3             NaN
Delta 4             0.0
Delta 5             0.0
Delta 6             0.0
Delta 7             0.0
Delta 8             NaN
Delta 9             0.993
Delta 10            0.0
Delta 11            0.0
Delta 12            0.0
Delta 13            1.0
Delta 14            0.0
```

Figure 37. Experiment 2 test 1.

Figure 37 shows the result of test 1. The predicate with the fault has the largest delta. In this test and all other test except one where the error was only introduced in the predicate the guilty predicate had the largest delta each times in experiment 3.

Between experiment 1 and 2 this statistical debugging technique showed promise from the perspective of finding bugs that exist directly in the predicates in a program. It was now time to see how this technique would perform when faults where introduced in assignment statement and other areas within a program.

2. Sensitivity Check

A sensitivity check was performed on experiment 1 to see how sensitive this debugging technique was at finding bug in other areas of a program other than predicates. Figure 38 show the results of the sensitivity check that was conducted.

	Current Predicates	Current Predicate or 1
observations=	50	50
true=	27	36
true/total=	0.54	0.72
1	**Assignment 1 or 2**	**Other 1 or 2**
observations=	50	50
true	25	28
true/total=	0.5	0.56

Figure 38. Sensitivity check.

The first test was to see how sensitive this technique was at indicating that the fault was in the predicate had the largest delta. Fifty tests were run in this case and 27 of the time the predicate that had the fault had the largest delta. Only fifty tests were run because each test had to have the error introduced manually. An error of commission was introduced and the test suite would run the *test program* 100,000 times to produce some quality statistics.

The second sensitivity test was to see how many times the guilty predicate would have the largest predicate or if the guilty predicate was one predicate up or down stream in the logical flow of the program. Just like in the previous case the error was only introduced in the predicate. In this case 50 tests were run and 36 times the conditions in this cases were satisfied which is 72 percent of the time. To quote Fred Brooks "there is no silver bullet" in testing and debugging, but this statistical debugging technique present an alternative tool that can be used during testing and debugging.

The third sensitivity check was to evaluate the efficiency of this technique in finding faults that exist in assignment statements that are either one or two predicates away from the predicate that is indicating it has the largest delta. As in the previous cases a total of 50 tests were conducted (manually inserting the faults) and 25 of the test indicated that the fault in the assignment was one or two predicates up or down stream in the logical flow of the program away from the predicated with the largest delta.

The fourth and final sensitivity check was to check the likelihood of finding errors that were 1 or 2 predicate up or down stream. Fifty tests were conducted, in 28 tests this techniques was able to indicate that the fault was 1 or 2 predicates away.

It has been concluded that this technique looses it ability to find a fault the further the fault is away from the predicate. In the fourth sensitivity check a fault was place inside a complex math function. The best that this technique could do was point to the math function. In larger program where functions can be hundreds of lines long, this statistical debugging technique is limited in what is can provide in the area of fault isolation.

In this research only errors of commission where addressed. They were the only ones that could be checked. The oracle program is just a copy of the test program with no errors introduced in it. By making these assumptions about the oracle, it must be understood that with an oracle errors of omission would be hard to find. Even by using an instrumentation tool for statistical debugging and having an oracle to monitor the behavior of predicates, there is a limit to debugging using this statistical debugging approach.

VI. CONCLUSION

A. SUMMARY

It has been concluded that the tool designed in this research to perform statistical debugging can be useful but it has limits. Faults in assignment statements may be associated with one or more predicates. Looking at a predicate with the largest delta may not lead you to the assignment statement with the fault, but will point to the fault in a lot of the time. When the assigned variable is used directly by a predicate then it has a stronger possibility of pointing to the guilty assignment statement

A variable's scope with-in a program is an important attribute to understand. A local variable with a fault was easier to pinpoint and had large deltas compared to variables that were not in the local scope of a predicate that was giving indications of a fault.

Faults located in mathematical computation within a method were hard to find. It was possible to trace back upstream from the predicate with the largest delta to the method with the fault, but this was the only indication given, the fault may exist anywhere within the method, this is where the statistical debugging technique utilized in this research looses it effectiveness.

Even though Statistical debugging is not exhaustive and will not find all faults, Statistical Debugging could be used to aid in the testing and debugging process because it will save time and resources as well as expedite the debugging process.

B. FUTURE WORK

There are several areas that can be researched in the future, but there are two areas that should receive the greatest attention.

First, what is the upper bound of statistical debugging (using an oracle approach) with respect to isolating faults. The fact that statistical debugging only monitors predicates means that it has limitations in what it can provide. This idea would be useful in the future because it would give some sense as to how useful this technique would be during testing and debugging.

Second, what if other profile data is gathered and used to help isolate faults. For example, what if the variable profile information is analyzed during a test run, is it possible that multiple profiles of different types can be used to enhance the ability of statistical debugging.

APPENDIX. SOURCE CODE

A. PRELIMINARY WORK

1. SortTest3.java

```
package thesis_work;

import java.util.*;

public class SortTest3 {

    public static void main(String arg[])
    {
        Random generator = new Random();
        Counter3 global1 = new Counter3(4);

        final int ARRAY_SIZE = 5;

        int i;
        int b;

        int x[];
        int y[] = null;

        int countNumbers;

        System.out.print("This is just a test program for my thesis
work, \n" + "my goal is to take in numbers from a user. Return
the\n" + "numbers in ascending order and print out the median.\n" +
"I will count the number of times the program iterates\n" +
                "through each looping statement as well!\n\n");

       //checking if arguments have been given or not at the command
      //line
        for(int k=0; k<5; k++) {

            if (arg.length==0){

                for(int j=0; j<ARRAY_SIZE; j++){

                    x[j] = generator.nextInt(500);
```

```
x = new int[ARRAY_SIZE];
                y = new int[ARRAY_SIZE];

                for(int j=0; j<ARRAY_SIZE; j++){

                    x[j] = generator.nextInt(500);

                //populating array y with the same numbers as array x.
                    y[j] = x[j];
                }


            }else
            {
                x = new int[arg.length]; //declaring an empty
array,initialized later

                for(i = 0; i< x.length; i++) //loop for putting the
values into the array
                    x[i]=Integer.parseInt(arg[i]);

            }


            countNumbers = x.length;

System.out.print("You entered " + countNumbers + " numbers to be " +
                    "sorted." + "\n\n");

            try //trapping any errors that might occur
            {

      //calling the method for sorting,defined later in the class
                selectionSort(x);

                //printing the sorted list
                System.out.println("\n\nHere is the sorted list:-");
```

```
for (i = 0; i < x.length; i++)
                        System.out.println(x[i] + "");

                  }
      //traps the error if specified arg is not a number
       catch (NumberFormatException E)                 {
System.out.println("The given argument/s is not a number...\n");
                  }

                  System.out.println("\n");

            int middle = x.length / 2; //subcript of middle element
            if (x.length % 2 == 1) {

                  //Odd number of elements -- return the middle one.
            System.out.println("Median is " + x[middle] + "\n");
                  } else {
                  // Even number -- return average of middle two
                  // Must cast the numbers to double before dividing.
      System.out.println("Median is " + (x[middle - 1] + x[middle])
                              / 2 + "\n");

                  }
            }
      }

      static void selectionSort(int x[]) //method for sorting the
list,made static for using in main
      {

            Counter3 global1 = new Counter3(4);

            int i, j;
            int pos = 0; //variable to fix a position in array
            int min = 0; //variable to fix a number as minimum.

            for (i = 0; global1.predicate(1, i < x.length - 1); i++)
{
```

```
                  min = x[i + 1]; //setting the minimum number
                  pos = i + 1; //fixing the position to the
index of that number

      for (j = i + 1; global1.predicate(2, j < x.length); j++)

                  {

                        if (global1.predicate(3, min > x[j]))
//if number at x[j] is graeter than minimum then

                        {

                        min = x[j]; //put x[j] in min
                        pos = j; //and change position to index
number of j

                        }
                  }

                  if (global1.predicate(4, x[pos] < x[i])) //if
number at x[pos] is less than number at x[i] then swap the
values

                  {

                        x[pos] = x[pos] + x[i];
                        x[i] = x[pos] - x[i];
                        x[pos] = x[pos] - x[i];
                  }

            }
            global1.print_array(4);

            /*if (i == x.length - 2){
            global1.print_array(4);
             }else {
                 //Do nothing!
             }  */
      }
}
```

2. Triangle.java

```
package thesis_work;

import java.util.Random;

public class Triangle {

      static Counter4 global1 = null;

      public static void main(String[] args) {

            global1 = new Counter4(11);

            Random generator = new Random();

            final int ARRAY_SIZE = 3;

            int x[];

            int a = 0;
            int l = 0;
            int b = 0;
            int m = 0;
            int c = 0;
            int n = 0;
            int atemp = 0;
            int ltemp = 0;
            int btemp = 0;
            int mtemp = 0;
            int ctemp = 0;
            int ntemp = 0;

            int current_profile[];
            current_profile = new int[4];

            int correct_profile[];
            correct_profile = new int[4];

            for (int k = 0; global1.predicate(1, k < 10000000); k++) {

                  if (args.length == 0) {

                        x = new int[ARRAY_SIZE];

                        for (int j = 0; global1.predicate(2, j <
ARRAY_SIZE); j++) {

                              x[j] = generator.nextInt(7);
                              a = x[0];
                              l = x[0];
                              b = x[1];
                              m = x[1];
                              c = x[2];
                              n = x[2];

                        }

                  } else {
                        x = new int[args.length]; //declaring an
empty array,initialized later

                        for (int i = 0; global1.predicate(3, i <
x.length); i++)
                  //loop for putting the values into the array
                              x[i] = Integer.parseInt(args[i]);

                        a = x[0];
                        l = x[0];
                        b = x[1];
                        m = x[1];
                        c = x[2];
                        n = x[2];

                  }

                  atemp = a * a;
                  btemp = b * b;
                  ctemp = c * c;
                  ltemp = l * l;
                  mtemp = m * m;
                  ntemp = n * n;

                  if (global1.predicate(4, a < b + c && b < a + c
&& c < a + b)) {
                        //System.out.println("This is a Triangle "
+ " " + a + " "  + b + " " +c + "\n\n");

                        current_profile[0] = 1;

                        if (global1.predicate(5, a == b && b == c))
{
      //System.out.println("This is a Equilateral Triangle\n\n");

                              correct_profile[1] = 1;
                              correct_profile[2] = 0;
                              correct_profile[3] = 0;
```

```
} else if (global1.predicate(6, a == c || a == b || b == c)) {
                              //System.out.println("This is an
Isosceles Triangle\n\n");

                              current_profile[2] = 1;
                              current_profile[1] = 0;
                              current_profile[3] = 0;

                        } else if (global1.predicate(7, atemp == b *
b + c * c
                                    || btemp == a * a + c * c ||
ctemp == a * a + b * b)) {


                              current_profile[3] = 1;
                              current_profile[2] = 0;
                              current_profile[1] = 0;

                        } else {


                        }

                  } else {
                        //System.out.println("This is not a Triangle"
+ " " + a + " "  + b + " " +c +"\n\n");

                        current_profile[0] = 0;
                        current_profile[1] = 0;
                        current_profile[2] = 0;
                        current_profile[3] = 0;
                  }

                  /*
                   * This portion of the triangle program is going to
represent a program
                   * that has an error it it.
                   */

                  //same as predicate 4
                  if (global1.predicate(8, l < m + n && m < l + n &&
n < l + m)) {


                        correct_profile[0] = 1;
```

```
//same as predicate 5
                        if (global1.predicate(9, l == m && m == n))
{
                              //System.out.println("This is a
Equilateral Triangle\n\n");

                              correct_profile[1] = 1;
                              correct_profile[2] = 0;
                              correct_profile[3] = 0;

                              //same as predicate 6
                        } else if (global1.predicate(10, l == n ||
l == m || m == n)) {
      //System.out.println("This is an Isosceles Triangle\n\n");
                              correct_profile[2] = 1;
                              correct_profile[1] = 0;
                              correct_profile[3] = 0;

                              //same as predicate 7
                        } else if (global1.predicate(11, ltemp == m
* m + n * n
                                    || mtemp == l * l + n * n ||
ntemp == l * l + m * m)) {
            //System.out.println("This is a Right Triangle\n\n");

                              correct_profile[3] = 1;
                              correct_profile[2] = 0;
                              correct_profile[1] = 0;

                        } else {

                        }

                  } else {

                        correct_profile[0] = 0;
                        correct_profile[1] = 0;
                        correct_profile[2] = 0;
                        correct_profile[3] = 0;

                  }

                  if (current_profile[0] == correct_profile[0]) {
                        global1.compare(0, 1 == 1);
                  } else {
                        global1.compare(0, 1 == 0);
                  }
```

```
if (current_profile[1] == correct_profile[1]) {
                        global1.compare(1, 1 == 1);
                  } else {
                        global1.compare(1, 1 == 0);
                  }

                  if (current_profile[2] == correct_profile[2]) {
                        global1.compare(2, 1 == 1);
                  } else {
                        global1.compare(2, 1 == 0);
                  }

                  if (current_profile[3] == correct_profile[3]) {
                        global1.compare(3, 1 == 1);
                  } else {
                        global1.compare(3, 1 == 0);
                  }

                  /*
                   * return both arrays back to zero
                   */

                  current_profile[0] = 0;
                  current_profile[1] = 0;
                  current_profile[2] = 0;
                  current_profile[3] = 0;

                  correct_profile[0] = 0;
                  correct_profile[1] = 0;
                  correct_profile[2] = 0;
                  correct_profile[3] = 0;

            }
            global1.print_array(11);

      }

}
```

B. BIG EXPERIMENT “EXPERIMENT 1”

1. SingleRunMinimum.java

```
////////////////////////////////////////////////////////////////////

//Program file name: SingleRun_Minimum.java

//© Tao Pang 2006

//Last modified: January 18, 2006

//(1) This Java program is part of the book, "An Introduction to
//Computational Physics, 2nd Edition," written by Tao Pang and
//published by Cambridge University Press on January 19, 2006.

//(2) No warranties, express or implied, are made for this program.

////////////////////////////////////////////////////////////////////

//An example of searching a minimum of a multivariable
//function through the steepest-descent method.

//(a) This program was taken from a java programming site, it was
augmented by
//Toriano A Murphy to be used in thesis research, altered 15 Oct 07.

package Thesis_Experiment_One;

import java.lang.*;
import java.util.Random;

//The original name of this class is "Minimum"
//File name is

public class SingleRun_Minimum {

      static Counter5 global = Counter5.getcounter5Object();
      static Random rand = new Random();

      public static void main(String argv[]) {
```

```
MinimumA minA = new MinimumA(null, 0, 0);
            MinimumB minB = new MinimumB(null, 0, 0);

            for (int k = 0; k < 100000; k++) {

                  double delA = 1e-6, aA = rand.nextDouble();
                  double delB = 1e-6, aB = aA;
                  double xA[] = new double[2];
                  double xB[] = new double[2];

                  xA[0] = 0.1;
                  xB[0] = 0.1;

                  xA[1] = -1;
                  xB[1] = -1;

                  minA.steepestDescent(xA, aA, delA);
                  minB.steepestDescent(xB, aB, delB);

                  //This is apart of the original program......
                  //System.out.println("The minimum is at"
                  //+ " x =  " + x[0] +", y =  " +x[1]);

                  //This method keeps track of the predicate
count for each run
                  global.sumvisitcorrect(6);

            }
//This method calculates the absolute difference between the
//same predicates
//in the oracle and the same predicates in the testing program.
            global.deltas(6);
      }

}
```

```
package Thesis_Experiment_One;


///////////////////////////////////////////////////////////////
//© Tao Pang 2006
//
//
//Last modified: January 18, 2006
//
//
//(1) This Java program is part of the book, "An Introduction to
//
//Computational Physics, 2nd Edition," written by Tao Pang and
//
//published by Cambridge University Press on January 19, 2006.
//
//
//(2) No warranties, express or implied, are made for this
program.
////////////////////////////////////////////////////////////////

//An example of searching a minimum of a multivariable
//function through the steepest-descent method.

//(a) This program was taken from a java programming site, it
//was utilized by
//Toriano A Murphy to be used in thesis research
//Last modified by Toriano: October 18, 2007.

import java.lang.*;


//This class will be the one where I make all the changes
//"MinimumB".  This is the
//testing class
public class MinimumB {

    static Counter5 global = Counter5.getcounter5Object();

    private double x[];
    private double a;
    private double del;

    //static Counter5 global2 = new Counter5(6);

    //Constructor
    public MinimumB(double MinBx[], double MinBa, double
MinBdel) {
```

```
x = MinBx;
        a = MinBa;
        del = MinBdel;

    }


//  Method to carry out the steepest-descent search.
    public void steepestDescent(double x[],double a, double del) {
        //System.out.println("MinimumB steepest Descent works");
        int n = x.length;
        double h = 1e-6;
        double g0 = g(x);
        double fi[] = new double[n];
           fi = f(x, h);
        double dg = 0;
        for (int i=0;global.predicateB(0, i<n);++i) dg +=
fi[i]*fi[i];
        dg = Math.sqrt(dg);
        double b = a/dg;

        while (global.predicateB(1, (dg > del))) {
            for (int i=0;global.predicateB(2, i<n);++i) x[i] -=
b*fi[i];
            h /= 2;
            fi = f(x, h);
            dg = 0;
            for (int i=0;global.predicateB(3, i<n);++i) dg +=
fi[i]*fi[i];
            dg = Math.sqrt(dg);
            b  = a/dg;
            double g1 = g(x);
            if(global.predicateB(4, (g1 > g0))) a /= 2;
            else g0 = g1;
        }
    }

//  Method to provide function f = gradient g(x).
    public static double[] f(double x[], double h) {
        int n = x.length;
        double z[] = new double[n];
        double y[] = (double[]) x.clone();
        double g0 = g(x);
        for (int i=0;global.predicateB(5, i<n);++i) {
            y[i] += h;
            z[i] = (g(y)-g0)/h;
        }
        return z;
    }
```

```
//  Method to provide function g(x).

    public static double g(double x[]) {
        return (x[0]-1)*(x[0]-1)*Math.exp(-x[1]*x[1])
        +x[1]*(x[1]+2)*Math.exp(-2*x[0]*x[0]);
    }

}
```

```
package Thesis_Experiment_One;


import java.lang.*;
import java.util.Random;

//MinimumA is the Oracle program, not changes will be made here!
public class MinimumA {

   //field needed for the constructor
    public double x[];
    public double a;
    public double del;

    static Counter5 global = Counter5.getcounter5Object();

   //constructor
    public MinimumA(double MinAx[], double MinAa, double MinAdel){

        x = MinAx;
        a = MinAa;
        del = MinAdel;
    }

    //Method to carry out the steepest-descent search.

    public void steepestDescent(double x[],double a, double del) {
        //System.out.println("MinimumA steepest Descent works");

        int n = x.length;
        double h = 1e-6;
        double g0 = g(x);
        double fi[] = new double[n];
        fi = f(x, h);
        double dg = 0;
        for (int i=0;global.predicate(0, i<n);++i) dg += fi[i]*fi[i];
        dg = Math.sqrt(dg);
        double b = a/dg;

        while (global.predicate(1, (dg > del))) {
            for (int i=0;global.predicate(2, i<n);++i) x[i] -=
b*fi[i];
            h /= 2;
            fi = f(x, h);
            dg = 0;
            for (int i=0;global.predicate(3, i<n);++i) dg +=
fi[i]*fi[i];
dg = Math.sqrt(dg);
            b  = a/dg;
            double g1 = g(x);
            if(global.predicate(4, (g1 > g0))) a /= 2;
            else g0 = g1;
        }
    }

//  Method to provide function f = gradient g(x).

    public static double[] f(double x[], double h) {
        int n = x.length;
        double z[] = new double[n];
        double y[] = (double[]) x.clone();
        double g0 = g(x);
        for (int i=0;global.predicate(5, i < n);++i) {
            y[i] += h;
            z[i] = (g(y)-g0)/h;
        }
        return z;
    }

//  Method to provide function g(x).

    public static double g(double x[]) {
        return (x[0]-1)*(x[0]-1)*Math.exp(-x[1]*x[1])
        +x[1]*(x[1]+2)*Math.exp(-2*x[0]*x[0]);
    }

}
```

```
package Thesis_Experiment_One;



/**
 * @author Toriano Murphy
 * This is the monitor class that acts as a global.  This Counter5 class
 * is a singleton class that is sharded by all the classes in this program.
 */
public class Counter5 {


      private static int numberofVisitA []; //for one run

      private static int numberofVisitB []; //for one run

      private static int numbertrueA []; //for one run

      private static int numbertrueB []; //for one run

      private static int numberofVisitCorrA_tot []; //for all runs

      private static int numberofVisitCorrB_tot []; //for all runs

      private static int numbertrueCorrA_tot []; //for all runs

      private static int numbertrueCorrB_tot []; //for all runs

      private static int numberofVisitCorrAB_tot [];//for all runs


      private Counter5(int num) {
            numberofVisitA =  new int[num + 1];
            numberofVisitB =  new int[num + 1];
            numbertrueA = new int[num + 1];
            numbertrueB = new int[num + 1];
            numberofVisitCorrA_tot =  new int[num + 1];
            numberofVisitCorrB_tot =  new int[num + 1];
            numbertrueCorrA_tot = new int[num + 1];
            numbertrueCorrB_tot = new int [num + 1];
            numberofVisitCorrAB_tot =new int [num +1];

            for (int i =0; i <= num; i++){
```

```
numberofVisitA  [i] =0;
                  numberofVisitB  [i] =0;
                  numberofVisitCorrA_tot [i]= 0;
                  numberofVisitCorrB_tot [i]= 0;
                  numbertrueCorrA_tot [i]= 0;
                  numbertrueCorrB_tot [i]= 0;
                  numbertrueA [i]= 0;
                  numbertrueB [i]= 0;
                  numberofVisitCorrAB_tot [i] =0;
            }
      }

      public static synchronized Counter5 getcounter5Object() {

            if (counter5Object == null){
                  //it's ok we can call this constructor
                  counter5Object = new Counter5(15);
            }
            return counter5Object;

      }

//    prevent the creation of another instance of this Counter5 
class
      public Object  clone()

      throws CloneNotSupportedException
      {
            throw new CloneNotSupportedException();

      }

      private static Counter5 counter5Object;

      public static boolean predicate (int n, boolean b){

            numberofVisitA[n]++ ;
            //System.out.println("numberofVisitA " + " " + n + " 
" +   numberofVisitA[n]);

            if (b){
                  numbertrueA[n]++;
                  //System.out.println("pred numbertrue  " + n + 
" " + numbertrueA[n]);
            }
            return b;
      }

      public static boolean predicateB (int n, boolean b){
```

```
numberofVisitB[n]++ ;


            if (b){
                  numbertrueB[n]++;
                              }
            return b;
      }


      public static void sumvisitcorrect(int num){

            for (int i =0; i< num; i++){


                  numbertrueCorrA_tot[i] += numbertrueA[i];
                  numbertrueCorrB_tot[i] += numbertrueB[i];

                  numberofVisitCorrA_tot[i] += numberofVisitA[i];
                  numberofVisitCorrB_tot[i] += numberofVisitB[i];
            }


            for (int j =0; j< num; j++){
                  numbertrueA[j] = 0;
                  numbertrueB[j] = 0;

                  numberofVisitA[j] = 0;
                  numberofVisitB[j] = 0;
            }
      }

      static void sumvisitincorrect(int num){

            for (int i =0; i< num; i++){
```

```
//System.out.println(numberofVisitB[i] + " " + i);
            }

            for (int j =0; j< num; j++){
                  numbertrueA[j] = 0;
                  numbertrueB[j] = 0;

                  numberofVisitA[j] = 0;
                  numberofVisitB[j] = 0;
            }
      }

      static void sumvisitincorrect(int num){

            for (int i =0; i< num; i++){


                  numbertrueA[i] = 0;
                  numbertrueB[i] = 0;

                  numberofVisitA[i] = 0;
                  numberofVisitB[i] = 0;

            }

      }
      static boolean increment (int i, boolean c){

            numberofVisitCorrA_tot [i]++;
//System.out.println("numberofVisit" +  i + "  " + numberofVisit[i]);

            if (c){
                  numbertrueCorrA_tot[i]++;
                  numbertrueCorrB_tot[i]++;

//System.out.println("numbertrueCorr  " + numbertrueCorr[i] + "\n");
            }else {
                  numbertrueCorrA_tot[i]++;
//System.out.println("numbertrueInCorr  " + numbertrueInCorr[i]
//+"\n");
```

```
}
            return c;
      }

      public static void deltas (int num) {

            for (int i =0; i < num; i++) {

                  System.out.println("Delta " + i + "\t\t" +

      (Math.abs((((float)numbertrueCorrA_tot[i]/(float)numberofVisitC
orrA_tot[i]))
                                          -

      (float)numbertrueCorrB_tot[i]/(float)numberofVisitCorrB_tot[i]
)));


            }

      }


      static void print_array(int num){

            System.out.println("Here is the statistical
data!!!!\n\n");
            for (int i =0; i <= num; i++) {

            }
      }
}
```

2. SingleRun_Experiment.java

```
package Thesis_Experiment_Two;

import java.util.Random;

import Thesis_Experiment_One.Counter5;

public class SingleRun_Experiment {

static Counter5 global = Counter5.getcounter5Object();
static Random rand = new Random();

public static void main(String argv[])  {

    Experiment2A exper2A = new Experiment2A(0, 0, 0);
    Experiment2B exper2B = new Experiment2B(0, 0, 0);



    for(int k=0; k < 100000; k++){

        double A = rand.nextDouble();
        double B = rand.nextDouble();
        double C = rand.nextDouble();

        exper2A.funmethod(A, B, C);
        exper2B.funmethod(A, B, C);



        global.sumvisitcorrect(15);

    }
    global.deltas(15);

 }
}
```

3. Experiment2B.java

```
package Thesis_Experiment_Two;

//This is the oracle class


import Thesis_Experiment_One.Counter5;



public class Experiment2A {

    public double a;
    public double b;
    public double c;
    static Counter5 global = Counter5.getcounter5Object();

    public Experiment2A(double ScattAa, double ScattAb, double
ScattAc){

        a = ScattAa;
        b = ScattAb;
        c = ScattAc;
    }

    public void funmethod(double a, double b, double c){

        if(global.predicate(0,(a==b ))){

            a = a*b;

        }else if(global.predicate(1, (a*a > 10))){

            for (int i=0;global.predicate(2, i < c);i++){

                i++;
            }

            while (global.predicate(3, (a < b))){

                a++;
                b--;
            }
        } if(global.predicate(4, (a < b))){

            a++;
            while (global.predicate(5,(a<b))){
                System.out.println("");
}
        }else if(global.predicate(6,(b < a))) {
            b++;

        }

        funmethod2();
        funmethod3();
    }

    public void funmethod2(){

        double chelsea = 0;
        double audrey  = 0;
        double tori    = 0;

        chelsea = a;
        audrey = b;
        tori = c;

        if(global.predicate(7,(chelsea < tori))){

            while (global.predicate(8,(chelsea < tori))){
                chelsea++;
                tori--;
            }

        }else if(global.predicate(9,(chelsea > tori + 10))){

            chelsea--;
            chelsea--;
        }
        for (int i =0;global.predicate(10, i< tori);i++){
               tori--;
               tori--;
               chelsea++;
               i++;
               audrey++;
        }
    }

    public void funmethod3(){

         b = a*a*a;

         while (global.predicate(11,(b < a))){
             a = a/1.5;
             b++;
         }

         if(global.predicate(12,(b < c))){
```

```
b++;
             c =c*3.14;

         }
         if(global.predicate(13,(c > 20))){

             c--;
             c--;
             c--;
             c = c/1.2;
         }
         if(global.predicate(14,(b < c))){
             b =b*1.7;

         }

    }

}
```

4. Experiment2B.java

```
package Thesis_Experiment_Two;

import Thesis_Experiment_One.Counter5;
import java.util.Random;

import Thesis_Experiment_One.Counter5;

//This is the test class where the changes will be made
public class Experiment2B {

    public double a;
    public double b;
    public double c;
    static Counter5 global = Counter5.getcounter5Object();

    public Experiment2B(double ScattBa, double ScattBb, double 
ScattBc){
        a = ScattBa;
        b = ScattBb;
        c = ScattBc;
    }

    public void funmethod(double a, double b, double c){

        if(global.predicateB(0,(a==b))){

            a = a*b;
        }else if(global.predicateB(1, (a*a > 10))){

            for (int i=0;global.predicateB(2, i < c);i++){

                i++;
            }
            while (global.predicateB(3, (a < b))){
                a++;
                b--;
            }
        } if(global.predicateB(4, (a < b))){

            a++;
            while (global.predicateB(5,(a<b))){
                System.out.println("");

            }
        }else if(global.predicateB(6,(b < a))) {
            b++;
        }

        funmethod2();
        funmethod3();
    }

    public void funmethod2(){

        double chelsea = 0;
        double audrey  = 0;
        double tori    = 0;

        chelsea = a;
        audrey = b;
        tori = c;

        if(global.predicateB(7,(chelsea < tori))){

            while (global.predicateB(8,(chelsea < tori))){
                chelsea++;
                tori--;
            }

        }else if(global.predicateB(9,(chelsea > tori + 10))){

            chelsea--;
            chelsea--;
        }
        for (int i =0;global.predicateB(10, i < tori);i++){
               tori--;
               tori--;
               chelsea++;
               i++;
               audrey++;
        }


    }

    public void funmethod3(){

         b = a*a*a;

         while (global.predicateB(11,(b < a))){
             a = a/1.5;
             b++;
         }

         if(global.predicateB(12,(b < c))){
             b++;
             c =c*3.14;
```

```
}
         if(global.predicateB(13,(c > 20))){

             c--;
             c--;
             c--;
             c = c/1.2;
         }
         if(global.predicateB(14,(b < c))){
             b =b*1.7;

         }

    }

}
```

C. INSTRUMENTATION TOOL

1. InstrumentationTool.java

```
package thesis_work;

/* @author Toriano Murphy
 * Thesis work in "Statistical Debugging"
 * Thesis advisor "Professor Makihal Auguston"
 * Co-Adviors "Richard Riehle
 */

//Copy text from an input file (1st file named on command
line) to an output file
//(2nd file named on command line) overlooking all comments.
The input file
//will be instrumented for use in statistical debugging
research.

import java.io.*;

class InstrumentationTool {

    public static void main(String[] args) throws IOException
{
        if (args.length < 2) {
            System.err.println("Please supply an input file
name and an output file name.");
            System.exit(1);
        }
        File inFile = new File(args[0]);
        File outFile = new File(args[1]);
        BufferedReader br = new BufferedReader(new
FileReader(inFile));
        BufferedWriter bw = new BufferedWriter(new
FileWriter(outFile));


        int ch;
        int tempch = 0;
        int Counter = 0;
        int Parenthesis = 0;
```

```
//main loop of the program
        while ((ch = br.read()) != -1) {
            if (ch == '/') {
                // here after seeing one slash; read the next
character
                // and decide what to do based on what it is
                ch = br.read();
                switch (ch) {
                case -1:
                    // end-of-file: if the character after the
"slash" is
                    // the end of file write out the slash and
exit the switch statement
                    bw.write('/');
                    break;
                case '/':
                    // if another slash -- read (and ignore)
chars until end of file
                    // or end of line.
                    bw.write('/');
                    bw.write('/');
                    ch = br.read();

                    //after seeing two slashes search for the end
of line
                    while (ch != -1 && ch != '\n') {
                        bw.write(ch);
                        ch = br.read();
                    }
                    if (ch == '\n' || ch == -1) //bw.write(ch);
                        break;

                case '*':
 // if the second character is an asterisk this is a block
//comment.
                    bw.write('/');
                    bw.write('*');
                    ch = br.read();
                    //ch2 = br.read();

// searches until the temch = * and ch = / everything thing else
//will be
                    // over looked since it is in the comment.
                    while (ch != -1  && ch != '/' ){
                        bw.write(ch);
                        tempch = ch;
                        ch = br.read();
```

```
}
 //This if statement checks to see if there is really a star slash
//(*/)
                    if(ch == '/' && tempch == '*'){
                        bw.write(ch);
                        ch = br.read();
                    }

                    break;

                    //default case if previous ones are not met
                default:
                    bw.write('/');
                bw.write(ch);
                ch = br.read();
                }
            }

            //check for keyword 'if' one character at a time
            if(ch == 'i'){

                ch = br.read();

                if(ch == 'f'){

                    //instrumentation for the "if" condition
                    Parenthesis++;
                    bw.write("if(global.predicate(");
                    //concerts the variable into a String
                    bw.write(String.valueOf(Counter));

                    Counter++;
                    bw.write(",");
                    ch =br.read();

                    //Search for the open parenthesis
                    while(ch != '('){

                        bw.write(ch);
                        ch = br.read();

                    }

                    bw.write(ch);
                    ch = br.read();
```

```
//Search for the last open parenthesis then add one more to
                    //complete the instrumentation
                    while(Parenthesis != 0){
                        if(ch == '('){
                            Parenthesis++;
                            bw.write(ch);
                            ch = br.read();
                        }

                        if (ch ==')'){

                            Parenthesis--;
                            bw.write(ch);
                            bw.write(ch);
                            ch = br.read();
                        }else {
                            bw.write(ch);
                            ch = br.read();
                        }
                    }

                    bw.write(")");


                }else{
                    bw.write("i");
                    bw.write(ch);
                    ch = br.read();
                }

            }

            //Instrumentation code for the "while condition

            if(ch == 'w'){
                ch = br.read();
                if (ch == 'h'){
                    ch = br.read();
                    if (ch == 'i'){
                        ch =br.read();
                        if (ch == 'l'){
                            ch = br.read();
                            if (ch == 'e'){
```

```
//instrumentation for the "while" loop
                                Parenthesis++;
                                bw.write("while
(global.predicate(");

bw.write(String.valueOf(Counter));

                                Counter++;
                                bw.write(",");
                                ch =br.read();

                                while(ch != '('){

                                    bw.write(ch);
                                    ch = br.read();
                                }

                                bw.write(ch);
                                ch = br.read();

                                while(Parenthesis != 0){
                                    if(ch == '('){
                                        Parenthesis++;
                                        bw.write(ch);
                                        ch = br.read();
                                    }

                                    if (ch ==')'){

                                        Parenthesis--;
                                        bw.write(ch);
                                        bw.write(ch);
                                        ch = br.read();
                                    }else {
                                        bw.write(ch);
                                        ch = br.read();
                                    }
                                }

                                bw.write(")");
                            }else{
                                bw.write("w");
                                bw.write("h");
                                bw.write("i");
                                bw.write("l");
                                bw.write(ch);
                                ch = br.read();
```

```
    }
                      }else {
                          bw.write("w");
                          bw.write("h");
                          bw.write("i");
                          bw.write(ch);
                          ch = br.read();
                      }
                  }else{
                      bw.write("w");
                      bw.write("h");
                      bw.write(ch);
                      ch = br.read();
                  }
              }else{
                  bw.write("w");
                  bw.write(ch);
                  ch = br.read();
              }
          }

          //instrumentation code for the "for condition
          if(ch == 'f'){

              ch = br.read();

              if (ch == 'o'){

                  ch = br.read();

                  if (ch == 'r'){

                      bw.write("for");
                      ch = br.read();

                      if (ch ==' ' || ch == '('){

                          while(ch != ';'){

                              bw.write(ch);
                              ch = br.read();
```

```
}

                            bw.write(ch);
                            bw.write("global.predicate(");

bw.write(String.valueOf(Counter));

                            Counter++;
                            bw.write(",");
                            ch = br.read();

                            while(ch != ';'){

                                bw.write(ch);
                                ch = br.read();

                            }
                            bw.write(")");
                            bw.write(ch);
                            ch = br.read();
                        }
                        else{

                            bw.write(ch);

                        }
                    }

                }else {bw.write("f"); bw.write(ch); }
            }

            else{bw.write(ch);}

        }
        br.close();
        bw.close();
    }
}
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] A.X. Zheng, M.I. Jordan, B. Liblit, A. Aiken, Statistical Debugging of Sampled Programs. Proceedings of the Seventh Annual Conference on Neural Information Processing Systems, 2003.

[2] B. Liblit, M. Naik, A.X. Zheng, A. Aiken, J.I. Michael, Scalable Statistical Bug Isolation. Proceeding of the 2005 ACM SIGPLAN. pp 15-26.

[3] A.X. Zheng, M.I. Jordan, B. Liblit, M. Naik, A. Aiken, Statistical Debugging: Simultaneous Identification of Multiple Bugs. Proceedings of the 23rd International Conference on Machine Learning. Pittsburg, PA, Vol 148. 2006. pp 1105-1112.

[4] B. Liblit, A. Aiken, A.X. Zheng, M.I. Jordan, Bug Isolation via Remote Program Sampling. ACM SIGPLAN, Vol 38, Issue 5. pp 141-154.

[5] C. Liu, Fault-aware Fingerprinting: Towards Mutualism between Failure Investigation and Statistical Debugging. ACM, 2006.

[6] J. Dennis, G. Neelam, Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction. IEEECS, 2007.

[7] Henry Lieberman, The Debugging Scandal And What To Do About It. Communications of the ACM. Vol 40, No. 4, April 1997.

[8] This link is to a repository of Java source code organized by topic. www.java2s.com, last accessed March 2008.

[9] L. Choa, F. Long, Y. Xifeng, H. Jiwaei, P.M. Samual, Statistical Debugging: A Hypothesis Testing-Based Approach. IEEE Transaction on Software Engineering, Vol.32, No. 10, October 2006. pp 831-848.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Dr. Mikhail Auguston
   Naval Postgraduate School
   Monterey, California

4. Richard Riehle
   Naval Postgraduate School
   Monterey, California

5. Toriano A. Murphy
   Naval Postgraduate School
   Monterey, California